Der Umgang mit dem Hoare-Kalkül zur Programmverifikation

1 Allgemeines

1.1 Ursprung

Schon in den frühen Zeiten der Programmierung trat das Bedürfnis auf, die entstandenen Artefakte auf deren Korrektheit hin analysieren zu können und sich nicht nur auf die Suche nach Fehlern beschränken zu müssen: Alan Turing hat sich dieser Fragestellung bereits 1949 gewidmet [Tur 49], fortgesetzt wurde dieser Gedanke dann im Wesentlichen von Floyd, dessen Artikel aus dem Jahr 1967 als eine Wurzel der Verifikation angesehen werden kann [Flo 67]. Hierauf aufbauend hat Hoare das 1969 vorgestellte System der axiomatischen Semantik entwickelt [Hoa 69].

Tatsächlich ist diese häufig auch als *Hoare-Kalkül* bezeichnete Methode der Verifikation also nicht ausschließlich das Werk von C.A.R Hoare. Die Regel zum Beweis der totalen Korrektheit von Schleifen geht nicht auf Hoare sondern auf eine lange Zeit nach Hoares Artikel veröffentlichte Arbeit von Dijkstra zurück [Dij82], die Regel für bedingte Anweisungen stammt von Lauer [Lau71].

Ein weiterer falscher Eindruck der leicht entstehen kann ist der, dass die Begriffe *Hoare-Kalkül* und *Verifikation* synonym zueinander verwendet werden können. Zwar wird in vielen Lehrveranstaltungen des Grundstudiums unter dem Begriff *Verifikation* lediglich die *Hoaresche Logik* behandelt, tatsächlich ist Verifikation mittlerweile ein recht umfangreiches Themengebiet, in dem der Hoare-Kalkül lediglich eine Methode unter vielen darstellt. Zahlreiche Spezialisierungen, die im Laufe der Jahre entwickelt wurden, z.B. für parallele, verteilte, nichtdeterministische oder auch disjunkte parallele Programme, dienen dem Zweck der Verifikation, werden aber über andere Formalismen, Axiome und Regelsysteme als ausschließlich der Hoarschen Logik realisiert. Ein Beispiel ist das *Model Checking*, eine Verifikationsmethode für reaktive Systeme, hierbei handelt es sich um Systeme, die mit ihrer Umgebung in Aktions-/Reaktionsbeziehungen treten, ein anderes die Programmanalyse, die sich im Gegensatz zum Hoare-Kalkül automatisieren lässt.

Die axiomatische Semantik im Hoarschen Sinne lehnt sich stark an Ideen an, wie sie am ehesten in der imperativen Sprache Pascal umgesetzt sind. Tatsächlich für Pascal entwickelt (auch der Eindruck entsteht aufgrund der inhaltlichen Nähe des Hoare-Kalküls zu den Konstrukten der Sprache Pascal gelegentlich unzutreffender Weise) ist der Hoare-Kalkül hingegen nicht. Hoares ursprüngliche Veröffentlichung *‚An axiomatic basis for computer programming'* stammt aus dem Jahr 1969. Der Entwickler der Sprache Pascal, Niklaus

Wirth, publiziert diese Sprache erstmals 1971 [Wir71]. Hoare hat seine Systematik 1973 mit Wirth zusammen unter einem ähnlich Titel *(An Axiomatic Definition of the Programming Language Pascal)* um einem Focus auf Pascal erweitert [HM 73].

Die Nähe der Hoarschen Logik zur Sprache Pascal resultiert aus der gemeinsamen Arbeit von Hoare und Wirth ab Mitte der 60er Jahre; einige Ideen von Wirth haben Hoare sicherlich bei der Entwicklung seiner Axiome inspiriert, der ebenfalls an Belangen der Semantik und Korrektheit interessierte Wirth hat aber im Gegenzug die Sprache Pascal auch entlang der Ideen von Hoare entworfen, offensichtlich so, dass sich zumindest die damals bekannte partielle Korrektheit damit zeigen ließ.

Neben zahlreicher Kritik an der axiomatischen Semantik ‚von außen' also an deren Anwendung schlechthin, an deren Kompliziertheit etc. – dies wird weiter hinten ausführlicher diskutiert – ist seit einer einschlägigen Publikation von Clarke 1979 auch ein schwererwiegender inhaltlicher Mangel bekannt: Sobald das Prozedurkonzept reichhaltig genutzt wird, ist es prinzipiell unmöglich, ein vollständiges Hoarsches Beweissystem zu finden [Cla 79]. Dies auch dann, wenn man sich auf logische Strukturen mit einem endlichen Datenbereich einschränkt. Damit kommt die Hoare'sche Logik in der Praxis relativ schnell an ihre Grenzen, denn kaum ein Softwaresystem, das heute realen Anforderungen genügen muss, kann aus Gründen der Komplexität in monolithischer Form erstellt werden.

1.2 Was ist der Zweck des Hoare-Kalküls?

Grundsätzlich besteht der Wunsch nach Software die fehlerfrei ist, die Erfahrung zeigt hingegen, dass bereits fehlerarme Software die Ausnahme darstellt und garantiert fehlerfreie sich bislang als nicht realisierbar erweist. Nach Faustregeln zur Häufigkeit von Fehlern ist beispielsweise bei einem Projekt wie Windows NT 5.0 mit einem Umfang von bis zu 27 Millionen Zeilen Code zunächst von rund 6 Millionen enthaltenen Fehlern auszugehen. Diese lassen sich durch Testen finden. Hierzu gibt es verschiedene Techniken wie Black Box- oder White-Box Tests, Anweisungs-, Bedingungs- und Pfadüberdeckungen und viele andere mehr. Je Testlauf – so besagt eine andere Faustregel – kann man damit rechnen, ca. 30 % der Fehler zu finden. Rund 20 Durchläufe *jedes* Systemteils führen dann langsam zu einer akzeptablen Fehlerquote, zu einem Zustand, in dem ein System dann zumindest vermarktet werden kann. Auch dies ein Unterfangen, das für unter 200 Millionen US-Dollar nicht realisierbar ist [Lov 98]. Man sieht an diesem Beispiel auch deutlich, dass wirkliche Fehler*freiheit* nicht im Entferntesten zu erwarten ist.

Testen vs. Verifikation:
- *Testen bedeutet das Suchen von Fehlern*
- *Verifikation will die Korrektheit eines Programms zeigen*

Ein Alternative zum Testen ist das Vermeiden von Fehlern bzw. das direkte Finden von Abweichungen zwischen Spezifikation und Implementation. Mit Testen lässt sich also immer nur ermitteln, wenn etwas nicht korrekt läuft, die bereits vorliegende Korrektheit läßt sich damit jedoch nicht zeigen. Wird beim Testen kein Fehler mehr gefunden, heißt dies auch keineswegs zwingend, daß ein Programm fehlerfrei ist, sondern nur, daß die gewählte Teststrategie keine Fehler mehr zutage fördert. Die Verifikation beansprucht für sich, direkt zu beweisen, daß ein Programm fehlerfrei ist.

Überprüfung von Software - Begriffe

Verifikation: Eine Methodik zum Durchführen eines formalen Beweises, der zeigen soll, dass ein Programm *P* eine Spezifikation *S* erfüllt.

Testen: Das Überprüfen eines Programms *P* mittels einer Testmenge *T*, ob P für *T* die Spezifikation *S* erfüllt.

Validierung: Überprüfung der Spezifikation *S*, ob diese die Anforderungen *A* tatsächlich erfüllt.

2 Voraussetzungen

Die Verifikation bedient sich häufig der Logik, genauer der Teilgebiete Aus*sagenlogik*, und *Prädikatenlogik*, daher gilt es sich damit vertraut zu machen, soweit dies nicht der Fall ist.

2.1 Aussagenlogik und Prädikate

Aussagenlogik beschäftigt sich mit Aussagen, Aussageformen und deren Verknüpfungen. Einige Aspekte der Aussagen- und Prädikatenlogik, die bei der Verifikation von Bedeutung sind, werden hier wiederholt. Die entsprechende einschlägige Literatur (z.B. Schöning, Logik für Informatiker [Sch 95]) kann und will dieser Abschnitt nicht ersetzen.

Aussagen

Eine Aussage nichts anderes als ein Satz, der (sinnvoll) mit *wahr* oder *falsch* bewertet werden kann.

Beispiele:

„Der Eiffelturm steht in München."
„Berlin ist pleite."
„Im Himmel ist Jahrmarkt."
„Die Zahl 7 ist prim"

Wichtig ist lediglich, dass sich eine Aussage formal als *wahr* oder *falsch* beantworten lässt, auch wenn, wie bei der dritten Aussage eine Bestätigung oder Verneinung von niemanden wirklich gegeben werden kann. Die beiden möglichen Beurteilungen einer Aussage *wahr* oder *falsch* werden als *Wahrheitswerte* bezeichnet. Im Gegensatz zu den obigen Beispielen sind folgende Sätze hingegen keine Aussagen:

„Wie spät ist es jetzt?"
„Komm her."
„Hey, Du da drüben."
„Hilfe!"

Keine Aussagen sind Fragen, Befehle, einzelne Worte sowie sinnlose Zeichenfolgen. Um mit den Aussagen einfacher arbeiten zu können, werden sie mit Variablen gleichgesetzt, die in der Logik meist Grossbuchstaben sind. Also:

A = „Der Eiffelturm steht in München."
B = „Berlin ist pleite."
C = „Im Himmel ist Jahrmarkt."

Die Aussagenlogik liefert nun Syntax und Semantik, um mit solchen Aussagenvariablen hantieren zu können. Hierfür gibt es logische Operatoren wie etwa Negation, Konjunktion und Disjunktion. Um zu sehen wie dies auf die Wahrheitswerte wirkt, berücksichtigt man alle Werte, die eine Aussagevariable annehmen kann und notiert das Ergebnis des Operators nach Anwendung auf die Aussagevariable.

Negation

Eine Verneinung bzw. Negation wendet den Sinn einer Aussage. Dies entspricht der Benutzung des Wortes *nicht*. Die Negation ist unär, d.h. sie hat nur einen Operanden.

A	$\neg A$
wahr	falsch
falsch	wahr

Eine Verneinung bzw. Negation wendet den Sinn einer Aussage. Dies entspricht der Benutzung des Wortes *nicht*. Die Negation ist unär, d.h. sie hat nur einen Operanden.

Konjunktion

Die Verundung bzw. Konjunktion verknüpft zwei Aussagen und liefert ein Ergebnis für beide Aussagen. Für die verknüpfte oder Gesamtaussage gilt hierbei, dass sie nur wahr sein, wenn beide Teile bereits wahr sind, in allen anderen Fällen resultiert für die Gesamtaussage ebenfalls der Wahrheitswert *falsch*. Die Konjunktion ist binär.

A	B	$A \wedge B$
wahr	wahr	wahr
falsch	wahr	falsch
wahr	falsch	falsch
falsch	falsch	falsch

Beispiel: Die Aussagen

A = „Wenn man über 18 ist, kann man einen Führerschein machen."
B = „Wenn man einen Sehtest besteht, kann man einen Führerschein machen."
lassen sich entsprechend der Konjunktion zu $A \wedge B$ verknüpfen:

$A \wedge B$ = „Wenn man über 18 ist und eine Sehtest besteht, kann man einen Führerschein machen."

Implikation

Die Folgerung bzw. Implikation verknüpft wiederum zwei Aussagen zu einer. In diesem Fall so, dass die Gesamtaussage auch dann bereits als wahr

gilt, wenn eine der beiden Aussagen A oder B bereits wahr ist. Die Implikation steht für die Beziehung, „*wenn ... dann ...*“

A	B	$A \rightarrow B$
wahr	wahr	wahr
falsch	wahr	wahr
wahr	falsch	falsch
falsch	falsch	wahr

Zu beachten sind die Festlegungen der Zeilen 2 und 4. Kann aus etwas Falschem etwas Wahres folgen? Aus logischer Sicht ja, aus falschen Daten kann zunächst immer alles folgen (das ist häufig das Verhängnisvolle). Wenn man falsch misst, rechnet oder eine Fehlerquelle in einem Versuchsaufbau hat, kann man zufällig immer auf ein (tatsächlich) richtiges Ergebnis oder ein falsches Ergebnis kommen. Die Wahl von *wahr* ist im Fall falscher Vorderglieder von Implikationen per Definition gefunden worden.

Beispiel: Die Aussagen

$A =$ „BILD schreibt immer die Wahrheit .“
$B =$ „BILD ist gut.“
lassen sich entsprechend der Implikation zu $A \rightarrow B$ verknüpfen:

$A \rightarrow B$ = „Wenn Bild immer die Wahrheit schreibt, dann folgt, dass Bild gut ist.“

Äquivalenz

Der Vergleich bzw. die Äquivalenz gleicht die Wahrheitswerte zweier Aussagen ab. Wenn beide Aussagen gleiche Wahrheitswerte haben (also auch wenn beide *falsch* sind) ist die Äquivalenz wahr.

A	B	$A \leftrightarrow B$
wahr	wahr	wahr
falsch	wahr	falsch
wahr	falsch	falsch
falsch	falsch	wahr

Beispiel: Die Aussagen

$A =$ „BILD schreibt die Wahrheit.“
$B =$ „BILD lügt nicht.“
lassen sich entsprechend der Äquivalenz zu $A \leftrightarrow B$ verknüpfen:

$A \leftrightarrow B$ = „Bild schreibt genau dann die Wahrheit, wenn sie nicht lügt.“

Exklusion (Antivalenz)

Die ausschließende Veroderung bzw. Exklusion stellt eine Veroderung dar, bei nur eine Teilaussage wahr sein darf, damit die Gesamtaussage wahr liefert (entweder nur A oder nur B, nicht beides und auch nicht keines von beiden). In unserer Alltagssprache wird das Wort *„oder“* sowohl für die Disjunktion als auch für die Exklusion verwendet.

A	B	$A \oplus B$
wahr	wahr	falsch
falsch	wahr	wahr
wahr	falsch	wahr
falsch	falsch	falsch

Beispiel: Die Aussagen

$A =$ „Man kann reinkommen.“
$B =$ „Man kann rauskommen.“
lassen sich entsprechend der Äquivalenz zu $A \oplus B$ verknüpfen:

$A \oplus B$ = „Man kann rein- oder rauskommen.“

Neben diesen gebräuchlichen Kombinationen der Belegung zweier Variablen mit Wahrheitswerten gibt es weitere, wie die Identität (unär), NAND oder NOR (binär), die hier nicht weiter betrachtet werden.

Häufiger von Bedeutung können noch die Formeln von de Morgan sein, die wichtige Beziehungen zwischen Aussagetermen wiedergeben:

(i) $\neg(A \wedge B) \Leftrightarrow \neg(A) \vee \neg(B)$
(ii) $\neg(A \vee B) \Leftrightarrow \neg(A) \wedge \neg(B)$

Um aussagenlogische Ausdrücke formulieren und auswerten zu können, muss eine Syntax vorliegen, die die Reihenfolge festlegt, in der zusammengesetzte Terme zu verarbeiten sind und Klammern aufgelöst werden müssen.

Mit der Aussagenlogik können also lediglich Aussagen gebildet und dann der Wahrheitswert bestimmt werden. Die Prädikatenlogik erweitert die Ausdrucksmöglichkeiten der Aussagenlogik, dies wird ergänzt um Prädikate bzw. Prädikatsymbole und um Quantoren.

Prädikate (Aussageformen)

Prädikate sind Sätze, die Variablen enthalten und die beim Ersetzen dieser Variablen mit Elementen einer gegebenen Menge eine Aussage bilden (also auf die Menge {wahr, falsch} abbilden).

Beispiele:

„Der Eiffelturm ist x Meter hoch“ (das heißt implizit: $x \in \mathbb{R}$)
„Die Zahl n ist prim.“ (implizit $n \in \mathbb{N}$)

Prädikate können von beliebiger Stelligkeit sein:

„Der *ggT* von i und j ist n.“ (implizit: $i, j, n \in \mathbb{N}$)
„Der x-te Kaiser von China heißt y.“ (implizit: $x \in \mathbb{N}$ und $y \in$ String)

Prädikatsymbole

Führt man für Prädikate *„Satzelement, Satzelement ... Satzelement“* eine abkürzende Schreibweise wie Folgende ein

$$Bezeichner(Argument_1, Argument_2, ..., Argument_n)$$

so nennt man *Bezeichner* Prädikatsymbol und den ganzen Ausdruck *atomare prädikatenlogische Formel.*

Beispiel:

ist_prim(*n*)
ggT(*i*, *j*, *n*)

Falls für ein Element $Argument_i$ aus einer Menge G

$$Bezeichner(Argument_1 \in G, ..., Argument_n \in G,) = \text{wahr}$$

gilt, so sagt man, *Bezeichner gilt für Argument.*

Beispiel:

ist_prim(*n*) gilt für 7.
ggT(*i*, *j*, *n*) gilt für 10, 25 und 5.

Prädikatsymbole können als Abbildungen von beliebigen Grundmengen auf boolesche Mengen interpretiert werden.

Beispiel:

$ist\_prim: \mathbb{N} \rightarrow$ ` mit $ist\_prim(n) = \begin{cases} \text{wahr} & \text{wenn } n \text{ eine Primzahl ist} \\ \text{falsch} & \text{sonst} \end{cases}$

$ggT: \mathbb{N}^3 \rightarrow$ ` mit $ggT(i, j, n) = \begin{cases} \text{wahr} & \text{wenn } n \text{ der } ggT \text{ von } i \text{ und } j \text{ ist} \\ \text{falsch} & \text{sonst} \end{cases}$

Damit fehlen als Ausdrucksmittel der Prädikatenlogik noch die *Quantoren*, zunächst der Begriff:

Quantoren

Der Begriff Quantor hängt mit dem Begriff der Quantität zusammen. Quantitativ möchte man in der Mathematik in der Regel zwei Dinge ausdrücken: *„Für alle n Elemente gilt“* und *„es gibt (mindestens) ein Element n, für das gilt“*. Den ersten Sachverhalt drückt man mit „$\forall n$“ aus, den zweiten mit „$\exists n$“.

$\exists n$ ... := *es gibt ein n ...*
$\forall n$ ... := *für alle n gilt ...*

Durch das Voranstellen des Allquantors $\forall$ vor einen Term entsteht eine *Allaussage*, durch das Voranstellen eines Existenzquantors $\exists$ eine *Existenzaussage*, dieser Vorgang wird auch als *Quantifizieren* bezeichnet. Es lassen sich weitere Quantoren realisieren wie etwa ein Keinquantor, diese sind in der Regel jedoch durch die Kombination aus Allquantor oder Existenzquantor mit einem logischen Operator ebenso erzeugbar.

Eine wichtige Unterscheidung besteht zwischen freien und gebundenen Variablen: eine gebundene Variable ist eine Variable die in Verbindung zu einem Quantor steht, eine freie Variable ist von einem Quantor unbeeinflusst.

Freie und gebundene Variablen

$\exists x.P(x, x)$	beide x sind durch die Bindung des x an den Quantor gebunden
$\forall x.P(x, y, z)$	x ist an den Quantor gebunden, y und z sind frei
$\forall(x, y, z).P(x, y, z)$	sowohl x als auch y und z sind gebunden, keine Variable von P ist frei

Besitzt eine Formel keine freien Variablen, nennt man sie *geschlossen*. Einer geschlossenen Formel kann man einen Wahrheitswert zuordnen.

Geschlossene Formel

2.2 Aufgaben

Aufgabe

Drücken Sie das Verhältnis $x + y = z$ in Form eines Prädikatsymbols aus.

Lösung:
Das Verhältnis $x + y = z$ in Form eines Prädikatsymbols ausdrücken können, erfordert eine Darstellung, die die drei Argumente x, y sowie z entgegen nimmt und per Definition ein boolsches Ergebnis zurückliefert.

$Sum: \mathbb{N}^3 \rightarrow \text{Bool}$ mit $Sum(x, y, z) = \begin{cases} \text{wahr wenn } z \text{ die Summe aus } x \text{ und } y \text{ ist} \\ \text{falsch sonst} \end{cases}$

Für 1,2 und 3 ergibt sich somit eine wahre Aussage, es gilt z.B.

$Sum(1, 2, 3) = \text{true}$

Ebenso gilt:

$Sum(11, 22, 33) = \text{true}$

Aber auch:

$Sum(1, 2, 4) = \text{false}$

da $1 + 2 \neq 4$ ist.

Aufgabe

Quantifizieren Sie die Formel $P(x)$

(i) mit dem Allquantor und
(ii) mit dem Existenzquantor.

Lösung:
(i) $\forall x.P(x)$
(ii) $\exists x.P(x)$

3 Die Verifikation nach Hoare

3.1 Aufbau von Hoare Regeln

Beim Hoare-Kalkül handelt es sich um bis eine Menge von Regeln, die sich aus Prämissen und Schlussfolgerungen zusammensetzen.

$$\begin{array}{c} \text{Prämisse}_1 \\ \text{Prämisse}_2 \\ \vdots \\ \text{Prämisse}_n \\ \hline \text{Konklusion} \end{array}$$

Der Strich unter den Prämissen (Voraussetzungen) ist eine weitere Art neben den Symbolen $\rightarrow$ und $\Rightarrow$ Folgerungen auszudrücken. Im Unterschied zu Formeln mit Folgerungen wie (A = B) und (B = C) $\Rightarrow$ (A = C) handelt es sich um *Schlussfolgerungen* (Konklusion, Conclusio). Dies ist anschaulich vergleichbar mit dem Strich unter dem Einkaufszettel: alle Elemente werden addiert, der Summenstrich trennt jedoch das Endergebnis von den anderen Summanden und Zwischenergebnissen, die man notieren könnte.

Beispiel:

P_1:	Es gelten die Pokerregeln
P_2:	Das Kartenspiel ist nicht gezinkt
P_3:	Wir ziehen die Pik 10
P_4:	Wir ziehen den Pik Buben
P_5:	Wir ziehen die Pik Dame
P_6:	Wir ziehen den Pik König
P_7:	Wir ziehen das Pik As
P_8:	Nur wir haben eine Strasse gezogen
P_9:	Wir behalten die Nerven

Schlussfolgerung: Diese Runde kann nur gewonnen werden

Aus syntaktischer bzw. algebraischer Sicht unterscheidet sich die Schlussfolgerung nicht wirklich von den bisher betrachteten Implikationen, die Unterscheidung mit dem Schlussstrich ist eher inhaltlich begründet. In verschiedenen Quellen wird auch auf diese Darstellung verzichtet und ausschließlich mit Pfeilsymbolen gearbeitet, hier jedoch wird auf sie zurückgegriffen, da sie anschaulicher erscheint.

Der Prozess des Schlussfolgerns auf diese Art ist auch als *deduktives Schliessen* bekannt. Grundsätzlich ist das nichts unbekanntes, wie das Beispiel zeigt, denken wir ohnehin in vergleichbarer Art und Weise, lediglich auf die formale Schreibweise verzichten die meisten Menschen im Alltag. Bestimmte Konstellationen aus Prämissen und Konklusionen, die in der Praxis häufiger

Modus tollens

zitiert werden, haben eigene Bezeichnungen. Die folgende Konstellation wird als *Modus tollens* bezeichnet:

Modus tollens

$$\begin{array}{c} A \Rightarrow B \\ \neg B \\ \hline \neg A \end{array}$$

Beispiel:

Mit einem Herz Ass, hätte ich eine Herz Strasse (P_1: $A \Rightarrow B$)
Ich habe keine Herz Strasse (P_2: $\neg$B)

Also habe ich kein Herz Ass bekommen (Konklusion: $\neg A$)

Nach Änderung der zweiten Prämisse zu A folgt ein gegenteiliger Schluss, dies wird *Modus ponens* genannt:

Modus ponens

$$\begin{array}{c} A \Rightarrow B \\ A \\ \hline B \end{array}$$

Modus ponens

Beispiel:

Mit einem Herz As, hätte ich eine Herz Strasse (P_1: $A \Rightarrow B$)
Ich ziehe ein Herz Ass (P_2: A)

Dann habe ich nun auch eine Herz Strasse (Konklusion: B)

Eine weitere bekannte Konstellation ist der *Modus barbara*, ein *Syllogismus*. Das heißt, es handelt sich um Aussagen, die von allen Elementen einer Menge auf einige bzw. mindestens eines schliessen. Der Moduls barbara ist wir folgt aufgebaut:

Modus barbara

$$\begin{array}{c} B \Rightarrow \text{C} \\ A \Rightarrow B \\ \hline A \Rightarrow \text{C} \end{array}$$

Modus barbara

Beispiel:

Alle Pokerspieler sind Betrüger	(P_1: $B \Rightarrow$ C)
Meine Freundin ist ein Pokerspieler	(P_2: $A \Rightarrow B$)
Meine Freundin ist ein Betrüger	(Konklusion: B)

3.2 Hoare-Tripel

Die Anwendung der Regeln des Hoare-Kalküls führt auf Hoare-Tripel, das heißt auf Gruppen von drei Elementen folgender Art:

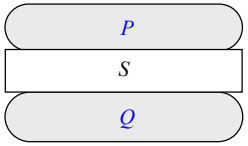
$$\{ P \}$$
S
$$\{ Q \}$$

Hierbei werden P und Q *Zusicherungen* genannt, S steht für eine Anweisung (ein Statement) in einer Programmiersprache. Weiterhin wird zwischen Zusicherungen unterschieden, die vor einer Anweisung S und danach stehen, P ist eine *Vorbedingung* und Q ist eine *Nachbedingung*. Der Zusammenhang zwischen P, S und Q ist folgender: Wenn vor der Ausführung von S die Programmvariablen das Prädikat P erfüllen, so erfüllen sie nach der Ausführung (und Terminierung) von S das Prädikat Q.

Weit verbreitet (weil im Quelltext realisierbar) ist die Notation der Hoare-Klauseln als Kommentar im Quelltext:

```
{ x + 1 > a }
x := x + 1;
{ x > a }
```

Alternativ zu der Notation in Quelltexten lässt sich auch eine an Struktogramme angelehnte Notation nutzen. Diese ist wesentlich übersichtlicher und bietet zudem den Vorteil, dass bei Schleifen und Verzweigungen vergleichbar Formularen ersichtlicht ist, wohin etwas geschrieben werden muss (das ist nicht immer offensichtlich, insbesondere wenn man sich in die Hoare-Technik noch einarbeitet).



Ein weiterer Vorteil dieser Darstellung ist, das sie nicht so fest an eine Programmiersprache gebunden ist, wie eine konkrete Implementierung. Ein Programm mit Hoare-Klauseln wird auch als Spezifikation bezeichnet.

3.3 Partielle und totale Korrektheit - Terminierung

totale Korrektheit:
Korrektheit + sichere Terminierung

Man unterscheidet generell zwischen *totaler Korrektheit* und *partieller Korrektheit,* wobei der Unterschied in der Terminierung des Programmstückes liegt. Schließt P die sichere Terminierung mit ein, spricht man bei erfolgtem Beweis von einem total korrekten Programmteil, andernfalls nur von partieller Korrektheit. Kann zunächst lediglich der Beweis partieller Korrektheit erbracht werden, gibt es die Möglichkeit, Terminierungsbeweise separat zu führen. In beiden Fällen kann gezeigt werden, daß der Quelltext korrekte Ergebnisse liefert. Im Fall totaler Korrektheit ist auch sicher gestellt, daß das Programm terminiert, partielle Korrektheit besagt hingegen nur, daß die Ergebnisse korrekt sind, *wenn* das Programm terminiert, nicht jedoch, dass es terminiert.

partielle Korrektheit:
Korrektheit ohne Beweis der tatsächlichen Terminierung

Die Grundlage für die Programmverifikation sind prädikatenlogische Ausdrücke, die auf den Zuständen eines Programms basieren, also mit den in den Variablen enthaltenen Werten an bestimmten Programmstellen. Diese Ausdrücke, Hoare-Klauseln genannt, basieren selbst wiederum auf sogenannten Beweisregeln.

3.4 Die Regeln nach C.A.R Hoare

Die Klauseln stehen zueinander in einem festen logischen Kontext – sie sind nicht einfach nur Notizen (dies ergäbe schließlich keinen Beweis im Sinne der Mathematik), sondern werden nach den bereits erwähnten Regeln von *unten nach oben* eingefügt[1]. Gelingt es, diese Klauseln in den Quelltext nach den Hoare-Regeln einzufügen ohne das sich widersprechende Terme auftreten und ohne, das diese in ihrer Semantik vom Quelltext abweichen, ist bewiesen, daß das Programm formal korrekt arbeitet, also für die definierten Eingaben die erwarteten bzw. korrekten Ausgaben liefert.

Der Hoare-Kalkül wird von unten nach oben angewandt.

R_0: Axiom der leeren Anweisung

$$\frac{}{\{ P \}\ \texttt{NOP}\ \{ P \}}$$

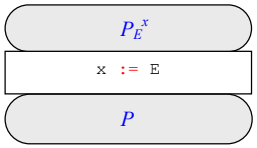
In einem Programm, daß keine Zustände verändert bzw. in dem nichts ausgeführt wird, bleibt P gültig.

R_1: Zuweisungen (*Zuweisungsaxiom*, *Regel für Ergibt-Anweisungen*)

$$\frac{}{\{ P_E^x \}\ \texttt{x := E}\ \{ P \}}$$

Hierbei bedeutet P_E^x, daß x durch E substituiert werden muß, damit die Nachbedingung P wahr wird. Da diese Regel keine Prämissen hat, wird sie auch als *Axiom* bezeichnet, also als Grundsatz, der nicht weiter zu begründen ist. In Verbindung mit dem Quelltext ergibt sich folgende Darstellung:

[1] Zwar könnte die Verifikation auch von oben nach unten vorgenommen werden, dies verursacht erfahrungsgemäß aber eine wesentlich kompliziertere Anwendung der Hoare-Regeln, daher wird hier auf diese Vorgehensweise verzichtet.

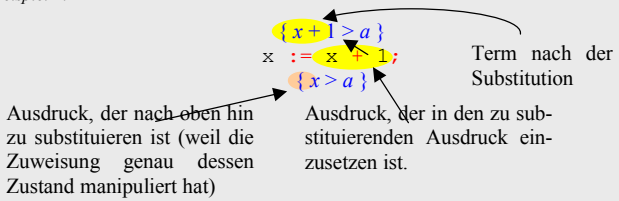


Die Regel wird herkömmlich[2] auf Zuweisungen so angewendet, daß die rechte Seite der Zuweisung im Quelltext jedes Vorkommen der Variablen in der oberen Klausel ersetzt. Also: Von unten kommend liegt folgendes Stück Quelltext vor, die untere Klausel ist dadurch gegeben, woher sie kommt, spielt in diesem Kontext keine weitere Rolle.

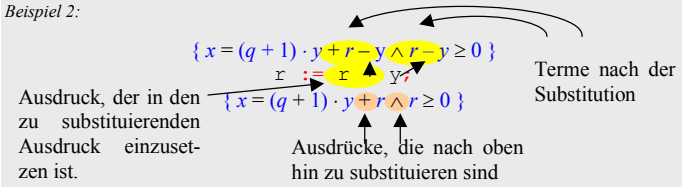
$\{ \quad ? \quad \}$
`x := x + 1;`
$\{ x > a \}$

Jedes Vorkommen des Ausdrucks x in der unteren Klausel muß in der oberen Klausel nun durch den rechten Teil der Zuweisung substituiert werden.

Beispiel 1:



Beispiel 2:



Hierbei können weitere Überlegungen notwendig sein. Denkbar ist zum Beispiel die Veränderung einer Relation anstelle der direkten Substitution mit dem Ausdruck `x + 1`:

$\{ x \geq a \}$
`x := x + 1;`
$\{ x > a \}$

Da der Hoare-Kalkül jedoch so weit wie möglich formal abgearbeitet werden soll, sind solche Umformungen nur hilfreich, wenn der vorliegende Term mit

[2] D.h. es gibt auch noch eine weitere Anwendung, die sog. Vorwärtsentwicklung, auf die hier nicht weiter eingegangen wird, da sie einerseits keine neuen Möglichkeiten eröffnet und anderseits komplizierter als die normale Rückwärtsentwicklung ist.

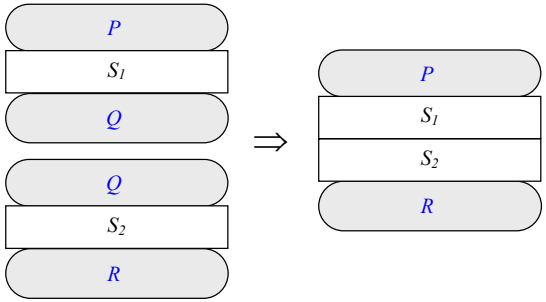
Termen in anderen Klausel zur Übereinstimmung gebracht werden soll. Jede Vereinfachung oder Zusammenfassung vorzunehmen, weil man sie gerade erkennt, ist genau genommen *nicht* Sinn der Sache. Es soll letztendlich bewiesen werden, daß ein Quelltext korrekt arbeitet und nicht der Grad der Virtuosität, in dem Beweisende Terme umformen kann.

R_2: Regel für Anweisungsfolgen, Regel der sequentiellen Komposition, Sequenzregel

$$\frac{\{ P \}\ S_1\ \{ Q \};\ \{ Q \}\ S_2\ \{ R \}}{\{ P \}\ S_1; S_2\ \{ R \}}$$

Für zwei Anweisungen S_1 und S_2, die nacheinander ausgeführt werden, können zu einem Programmstück S bestehend aus der Sequenz $S_1; S_2$ zusammengesetzt werden, wenn die Nachbedingung von S_1 mit der Vorbedingung von S_2 identisch ist.

Mathematisch ausgedrückt: Wenn S_1 Variablen von einem Zustand σ nach den Vorbedingungen P aus in einen Zustand σ' versetzt werden, in dem die Nachbedingungen Q gültig sind, und S_2 von einem Q-Zustand σ' in einen R-Zustand σ'' führt, dann führt die Nacheinandderausführung von S_1 und S_2 auch von einem Zustand σ zu einem Folgezustand σ''.



Die Verifikation funktioniert anhand von logischen Ausdrücken, sogenannten Zusicherungen, die um die Programmteile herum angebracht werden. Der Ansatz bei dieser Technik sind die *Zustände* eines Programms - also die jeweiligen Werte der enthaltenen Variablen. Die Folge einfacher Anweisungen

```
r := r - y;
q := q + 1;
```

Hier kann gemäß der Regel für Anweisungsfolgen zeilenweise vorgegangen werden:

$\{ x = y \cdot (q + 1) + r - y \}$
`r := r - y;`
$\{ x = y \cdot (q + 1) + r \}$
`q := q + 1;`
$\{ x = y \cdot q + r \}$

... oder umfassender:

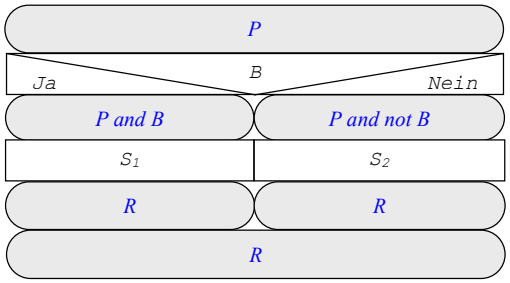
$$\{ x = y \cdot (q + 1) + r - y \}$$
```
r := r - y;
q := q + 1;
```
$$\{ x = y \cdot q + r \}$$

Die Regel der sequentiellen Komposition wird nur an einer Stelle direkt angewandt, sie schafft die Grundlage dafür, daß Anweisungsfolgen in der gezeigten Art nacheinander überhaupt *abverifiziert* werden können: Geht man von der zweiten Spezifikation aus, die nur umfassend verifiziert ist, erlaubt die Regel in dieser Hinsicht betrachtet die im darüber stehenden Listing gezeigte Einbringung von Hoare-Klauseln zwischen den Quelltextzeilen.

R_{3a}: Regel der Fallunterscheidung, Regel der bedingten Anweisung

$$\frac{\{ P \wedge B \}\ S_1\ \{ R \} \qquad \{ P \wedge \neg B \}\ S_2\ \{ R \}}{\{ P \}\ \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{ R \}}$$

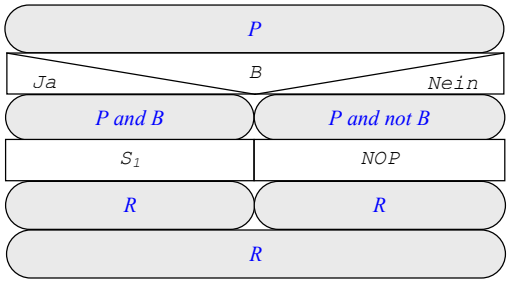
S_1 führt unter der Bedingung B von einem Zustand P auf einen Zustand R, ebenso S_2 unter der Bedingung $\neg B$. In der Struktogramm-Darstellung:



R_{3b}: Regel der Fallunterscheidung, Regel der bedingten Anweisung
Diese Variante der Regel *3a* dient der Verifikation von Verzweigungen ohne Alternative (den Else-Teil).

$$\frac{\{ P \wedge B \}\ S\ \{ R \} \qquad P \wedge \neg B \Rightarrow R}{\{ P \}\ \texttt{if}\ B\ \texttt{then}\ S\ \{ R \}}$$

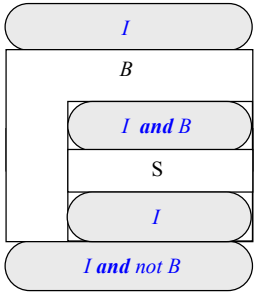
Allgemein kann man hier die Regel *3a* anwenden und den Quelltext unter S_2 als *NOP* (no Operation, leer) ansehen.



R_{4a}: Regel der Iteration (für präkonditionierte Schleifen)

$$\frac{\{ I \wedge B \}\ S\ \{ I \}}{\{ I \}\ \texttt{while}\ B\ \texttt{do}\ S\ \{ I \wedge \neg B \}}$$

Bei While-Schleifen wird der Rumpf S der Wiederholungsanweisung so lange wiederholt, bis die Wiederholungsbedingung B nicht mehr erfüllt ist (also bis $\neg B$ gilt). Zur Verifikation von While-Schleifen ist es notwendig, eine sogenannte *Invariante I* zu finden. Invarianten gelten nach jedem Schleifendurchlauf und beschreiben innerhalb der Schleifendynamik das Gleichbleibende. Das Finden von Invarianten ist eine der Schwierigkeiten des Hoare-Kalküls, mit Erfahrung und Übung werden Invarianten einfacher und schneller gefunden, einen eindeutigen und sicheren Weg hierfür gibt es jedoch nicht.[3]



Allerdings gibt es schon einige Heuristiken, d.h. Faustregeln, die eine ganze Reihe von Fällen bereits abdecken.

1. Eine Konstante variabel machen
2. Den Bereich einer Variablen erweitern
3. Eine Disjunktion hinzufügen (A um B zu A $\vee$ B ergänzen)

[3] Das Problem ist ähnlich dem des Programmierens selbst – auch hierbei handelt es sich um eine intelligente Tätigkeit, die zwar erlernt und vermittelt werden kann, für die selbst aber keine formal eindeutige Anleitung gegeben werden kann.

4. Eine Konjunktion aus einem Term weglassen, etwa wenn dieser Teil der Schleifenbedingung entspricht
5. Die im Verlauf der Schleife manipulierten Variablen in einer Tabelle mit einer ausreichenden Anzahl von Werten durchtracen um die Gesetzmäßigkeit leichter zu erkennen, die sich hinter dem Ablauf verbirgt.
6. In vielen Fällen (bei Berechnungen nach mathematischen Vorgaben) erfüllt die Rechenregel, die ohnehin Motivation für die Entwicklung der Schleife war, bereits die Anforderungen, die an eine Invariante gestellt werden (also eben invariant zu sein).

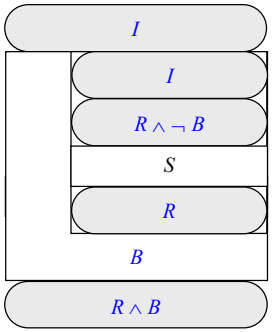
Im Allgemeinen gibt es mehrere mehr oder weniger gleich gute Schleifeninvarianten; falls man ungeschickt wählt, hat man eventuell mehr Schreibarbeit, es ist aber kein Fehler. Alle Varianten sind im hinteren Teil mit den Beispielen gezeigt.

Neben der Korrektheit von Schleifen muß in separaten Terminierungsbeweisen auch die Terminierung von Schleifen bewiesen werden, wenn die *totale* Korrektheit von Programmen gezeigt werden soll, andernfalls liegt nur eine *partielle* Korrektheit vor.

R_{4b}: Regel der Iteration (für postkonditionierte Schleifen)
Die Anwendung dieser Regel ist nur bedingt empfehlenswert. Generell sind Schleifen gleichmächtig, keine der drei Schleifen (in Pascal While-, For- und Repeat-Schleifen, allgemein Schleifen mit Eintrittsbedingung, Zählschleifen, Schleifen mit Austrittsbedingung) ermöglicht Implementierungen, die sich mit den jeweils anderen Konstrukten nicht erreichen läßt. Da die Verifikation der Schleife mit Eintrittsbedingung einfacher ist als die der anderen Schleifen, ist es üblich und wird empfohlen, für diesen Zweck die Schleife im Quelltext als Schleife mit Eintrittsbedingung zu formulieren. Die Hoare-Regel für Schleifen mit Austrittsbedingung:

$$\frac{\{ I \}\ S\ \{ R \} \qquad \{ R \wedge \neg B \}\ S\ \{ R \}}{\{ I \}\ \texttt{repeat}\ S\ \texttt{until}\ B\ \{ R \wedge B \}}$$

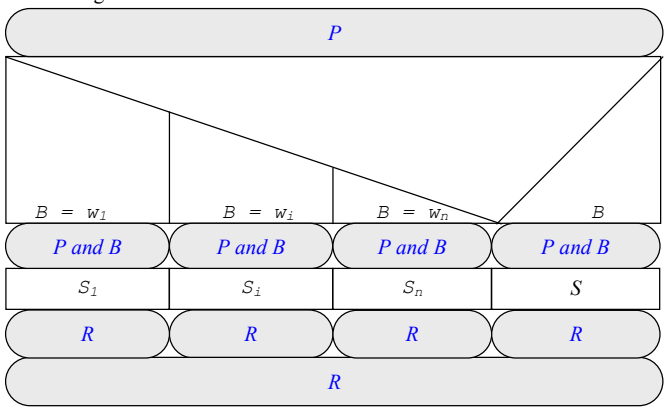
In Struktogramm-Notation:



R_5: Selektive Anweisung

$$\frac{\{ P \wedge (B = w_i) \}\ S_i\ \{ R \} \qquad (i = 1\ (1)\ n)}{\{ P \}\ \texttt{case}\ B\ \texttt{of}\ w_1: S_1; \ldots; w_n: S_n; \{ R \}}$$

In Struktogramm-Notation:



R_{6a}: Implikationsregel

$$\frac{\{ P \}\ S\ \{ Q \},\ Q \Rightarrow R}{\{ P \}\ S\ \{ R \}}$$

Wenn über einen Quelltext S aus einem Zustand P ein Folgezustand Q resultiert und aus Q wiederum ein Zustand R, dann folgt R letzten Endes bereits aus den Zustandsmanipulationen die der Quelltext S im laufenden Programm verursacht.

R_{6b}: Implikationsregel

$$\frac{P \Rightarrow R,\ \{ R \}\ S\ \{ Q \}}{\{ P \}\ S\ \{ Q \}}$$

Wenn aus einem Zustand P ein Zustand R folgt, aus dem über den Quelltext S ein Zustand Q angenommen wird, dann folgt Q bereits aus P.
Dies ist das Handwerkszeug, hiernach wird nun in diversen Beispielen gezeigt, wie die Verifikation damit durchgeführt wird.

3.5 Vorgehen beim Verifizieren

Der Beweis der Korrektheit eines Programmes oder Programmteiles wird nun in den folgenden Schritten vollzogen:

1. Die Nachbedingungen des gesamten Programmstückes werden notiert. Es wird also niedergeschrieben, zu was das Programmstück eigentlich dient. Hieran muss sich dann der Quelltext messen lassen: einzig an dem, was das

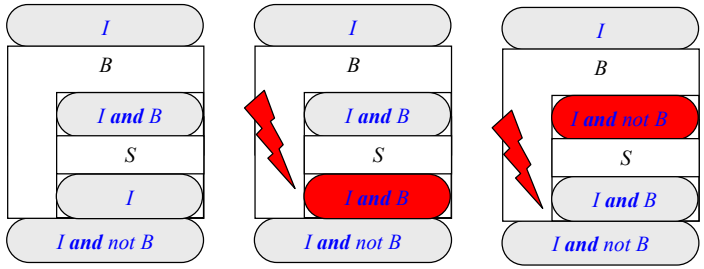
Programm machen soll. Die letzte Anweisung, die ausführbaren Code enthält, wird ermittelt (also kein „end“ oder auch keine anschließende Rückgabe von Werten – obwohl diese auch einer Zuweisung entsprechen kann, siehe Beispiel). Hier hinter wird die Nachbedingung eingetragen.
Sofern eine algebraische Spezifikation oder eine andere formale Beschreibung vorliegt, kann die Nachbedingung auch hier entnommen werden. Sofern diese Beschreibung noch gültig ist, muss das Programm das ermitteln, was laut Spezifikation vorgegeben ist.

2. Die Vorbedingungen werden notiert. Diese folgen ähnlich den Nachbedingungen aus dem Kontext und können „schwach“ gehalten werden. Wenn einem hierbei nichts einfällt, kann theoretisch *false* als Vorbedingung eingetragen werden. Dies ist als Vorbedingung allerdings nicht nur schwach sondern auch trivial, da aus etwas Falschem immer alles gefolgert werden kann (eine Eigenschaft der Implikation, siehe vorne Abschnitt *Aussagenlogik und Prädikate*).
3. Von unten nach oben werden nun die passenden Hoare-Regeln für die jeweiligen Anweisung auf den gesamten Quelltext angewendet. Schleifen werden zunächst von außen behandelt, dann folgt der Rumpf, wobei hier wieder von unten nach oben vorgegangen wird. Sind die inneren Anweisungen mit Klauseln versehen (und ist die Schleife auch innen mit Klauseln versehen, folgt die nächste Anweisung oberhalb der Schleife.
4. Wenn alle Anweisungen mit Hoare-Klauseln versehen sind, liegt eine *Beweisskizze* oder *Spezifikation* (auch Tableau, z.B. in [Bac 89]) vor. Taucht hierbei kein Widerspruch auf, zeigt dies, dass ein Programmteil *S* unter der Vorbedingung *P* auf die Nachbedingung *Q* führt und der Beweis ist erbracht.

Beweisrichtung und Richtung des Hoare-Kalküls

Die Literatur äußert sich verschiedentlich zur „Richtung“ des Hoare-Kalküls und schreibt beispielsweise, dieser sei rückwärts gewandt. Das ist in bestimmten Phasen richtig, in anderen jedoch absolut nicht.
Beim *Finden* von Hoare-Klauseln (also beim Erstellen der Beweisskizze) ist es zutreffend, dass diese von unten nach oben *ermittelt* werden. Die Hoare-Regeln bieten diese Nutzung an (von oben nach unten zu nutzende Regeln können zwar aus den vorhandenen abgeleitet werden, sind aber wesentlich umständlicher und schwerer zu nutzen), ferner erscheint es sinnvoll nicht in der selben Richtung zu verifizieren, in die man bereits beim Implementieren falsch denkt. Die Wahrscheinlich Fehler zu finden und nicht in den Beweisen fortzupflanzen ist dabei größer.

Fehler können beispielsweise folgender Art sein:



Beispiel 3, Fehler in Schleifen: 1) korrekte Spezifikation, 2) mögl. Endlosschleife, 3) den Inhalt ignorierende Schleife.

Die Verifikation ist abgeschlossen, wenn man von den letzten Anweisungen bzw. von den Zielvorgaben bis hin zum Anfang das Programm durchgegangen ist, ohne dass Widersprüche in den Klauseln aufgetreten sind.

3.6 Beispiele für Verifikationen

3.6.1 Ermittlung des Restes der Ganzzahldivision

Das folgende Stück Code ermittelt den Rest einer ganzzahligen Division mit Eingaben für x und $y \geq 0$. Teilt man etwa 25 durch 4 mit dem Aufruf *rest(25, 4)* paßt dies 6 mal ($6 \cdot 4 = 24$), der Rest 1 bleibt übrig. Diese Reste ermittelt die folgende Funktion:

```
function rest (x : integer; y : integer) : integer;
var q, r : integer;
begin
  q := 0;
  r := x;
  while r >= y do
  begin
    r := r - y;
    q := q + 1;
  end;
   rest := r;
end;
```

Das Prinzip ist, solange von der größeren Zahl den Divisor abzuziehen, bis das Ergebnis kleiner als der Divisor ist, dies ist der Rest.

Der ausführbare Code dieses Programmes bzw. für die Verifikation relevante Teil ist grau abgesetzt, obwohl die Funktion insgesamt 12 Zeilen umfaßt, werden im weiteren nur fünf Zeilen betrachtet (gelb bzw. orange herausgestellt). Die restlichen Zeilen greifen nicht mehr in die Zustände der Variablen zur Laufzeit ein und sind daher für die Verifikation ohne Bedeutung. Betrachtet werden also lediglich folgende Abschnitte:

```
q := 0;
r := x;
while r >= y do
   r := r - y;
   q := q + 1;
```

Hierin umfasst die While-Schleife die unteren Anweisungen und stellt somit die unterste äußere Anweisung (orange) dar, die beiden Zuweisungen ‚darunter' stellen den Schleifeninhalt dar und werden daher danach betrachtet.

Schritt 1 – Aufstellen der Vor- und Nachbedingung:

Für die Vorbedingung P gilt: Es werden gültige, d. h. sinnvolle, Eingaben vorausgesetzt. In diesem Beispiel wird zur Bedingung gemacht, dass x größer oder größer gleich 0 ist und dass y größer 0 sein soll, also

$$x \geq 0 \wedge y > 0$$

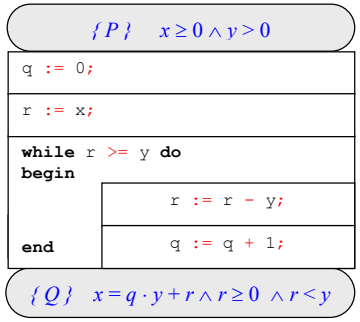
Für die Nachbedingung R gilt: Es gibt ein x, welches durch y geteilt werden soll. Als Ergebnis interessiert jedoch nicht, wie oft diese Division aufgeht also ein z aus $z = x / y$, sondern wieviel der Rest, der nicht teilbar ist beträgt. Hierzu wird in q Durchläufen der Betrag y von x abgezogen, es gilt also (dies ist nichts weiter als die Umkehrung der Definition der Division mit Rest):

$$x = q \cdot y + r$$

Dies geht solange, bis nur noch ein Rest r, der kleiner als y sein muß, übrig bleibt. Weitere Bedingungen sind also

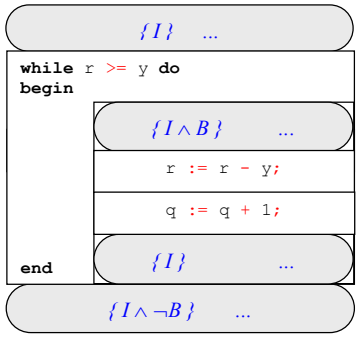
$$r \geq 0 \text{ und } r < y$$

Es ergibt sich damit folgendes umfassendes Hoare-Tripel (wenn man den Quelltext als ein umfassendes Element betrachtet):



Schritt 2 – Verifikation der letzten äußeren Anweisung (der While-Schleife)

Zunächst können gemäß Iterationsregel die formalen Zustandsklauseln vermerkt werden:



Iterationsregel:
- $\{ I \}$
- $\{ I \wedge \neg B \}$
- $\{ I \wedge B \}$

müssen gefunden werden

Dann müssen die Parameter für die Klauseln genauer bestimmt werden. Am einfachsten läßt sich hierbei B finden, die Bedingung B, die für die Klausel gesucht wird, ist immer identisch mit der Klausel, die direkt in der While-Schleife benutzt wird, sie läßt sie in der Regel abschreiben.

$$B\text{: } r \geq y$$

Ebenso einfach ist die (Gegen-) Bedingung $\neg B$ zu finden, die in der Klausel hinter der Schleife stehen muß – durch schlichte Verneinung von B.

$$\neg B\text{: } \neg (r \geq y)$$

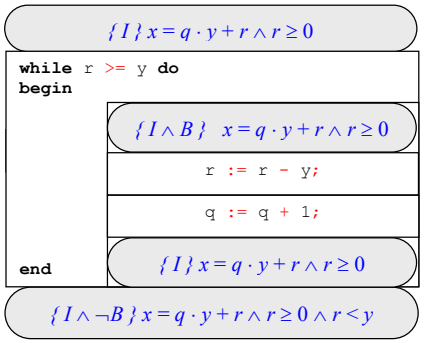
Dem Sinn nach also $r < y$, eine Umformung, die sich zwar anbietet aber nicht gefordert (und hier auch nicht hilfreich[4]) ist.

Anspruchsvoll und intelligent zu lösen ist die Findung der Invarianten I. Als allgemeine Ansätze empfohlen werden kann die Untersuchung der Zusammenhänge der Zustände, die die Variablen annehmen können, etwa durch Eintragen in Tabellen mit einer kleinen Anzahl von Werten (siehe folgende Beispiele). Oft führt auch das Übernehmen nachfolgender Klauseln wie der Nachbedingung zum Ziel. Tatsächlich gilt die Nachbedingung an allen Stellen, an denen I gefordert ist und stellt keine triviale Anforderung dar. Für die Invariante gilt also:

$$I\text{: } x = q \cdot y + r \wedge r \geq 0$$

[4] Schließlich soll der Hoare-Kalkül formal arbeiten und nicht auf derartige intelligente Überlegungen und Ideen basieren.

Damit ergibt sich für die Schleife:



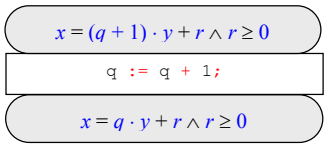
Schritt 3:

Da hier eine Schleife vorliegt sind zunächst die Anweisungen innerhalb dieser Schleife zu suchen, hier widerrum von außen nach innen und von unten nach oben. Im konkreten Fall gibt es keine weitere innere Ebene (Schleife oder Verzweigung), so daß mit der unteren Zuweisung fortgefahren wird.

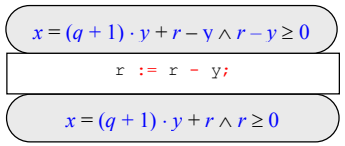
Die Nachbedingung des Zuweisungsaxion wird aus der Zeile der Schleifen-Verifikation übernommen. Generell ist die Nachbedingung einer Zuweisung eine vorstehende bzw. von unten zu übernehmende Klausel, also

i) eine Invariante,
ii) die Nachbedingung einer Verzweigung,
iii) die Nachbedingung des gesamten Quelltextteils oder
iv) die Vorbedingung einer nachstehenden Zuweisung.

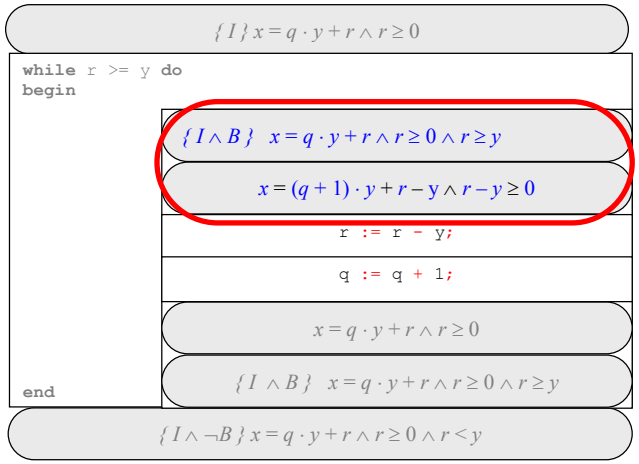
Der vorhergehende Zustand P_E^x wird gemäß Zuweisungsaxiom durch einfaches Substituieren des betreffenden Ausdrucks (hier die Variable q) mit dem rechten Teil de Zuweisung in der Klausel ermittelt. Es wird also in der Nachbedingung q jedes Vorkommen durch den q zugewiesenen Ausdruck (hier also $q + 1$) ersetzt.



Ebenso wird mit der zweiten Zuweisung verfahren, wobei deren Nachbedingung wiederum die aus dem vergangenen Schritt hervorgegangene Klausel ist.



Hier entsteht nun die Situation zwei aufeinander treffender Klauseln, die Invariante in Verbindung mit der Schleifenbedingung und das nun nach oben hervorgearbeitete Ergebnis der Zuweisungen – hierbei sind nun Ausdrücke verschiedener Gestalt hervorgetreten:



Hierbei ist zu ermitteln, ob die Ausdrücke nur verschieden aussehen, oder inhaltlich übereinstimmen (erst eine inhaltliche Abweichung stellt einen Widerspruch dar). Hier werden die Terme vor und nach der Adjunktion separat betrachtet, sind sie äquivalent, behält auch die Adjunktion ihre Gültigkeit:

Vorderer Teil:

$$\begin{aligned} x &= (q+1)\cdot y + r - y \\ &= q\cdot y + y + r - y \\ &= q\cdot y + r \end{aligned}$$

Hinterer Teil

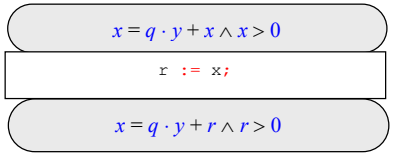
$$\begin{aligned} & r - y \geq 0 \;/\; + y \\ \Leftrightarrow\; & r \geq y \end{aligned}$$

Beide Teile der Adjunktion sind äquivalent und somit auch beide Klauseln.

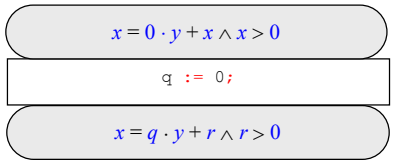
Schritt 4:

In derselben Weise werden nun die beiden letzten Anweisungen behandelt. Die vorhergehende Klausel für die erste Zuweisung vor der While-

Schleife stellt hierbei erneut die für die Schleife aufgestellte Invariante dar. Hierin wird die Variable *r* durch die Variable *x* substituiert.



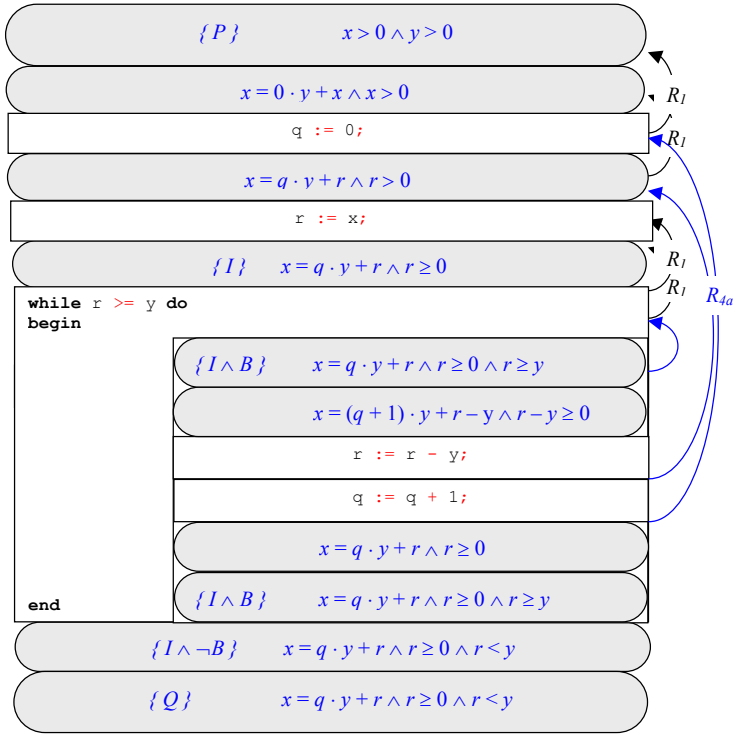
Ebenso wird auch mit der Initialisierung der Variablen *q* verfahren:



Schritt 5:

Abschließend entsteht auch aus der Vorbedingung des Programms und der oberen Klausel aus der ersten Zuweisung kein Widerspruch.

Aus dem Quelltext und der Hoare-Klauseln geht die folgende Spezifikation (auch als Beweisskizze bezeichnet) hervor:



3.6.2 Bilden des Quadrats aus einer Zahl

Das folgende Programm bildet aus einer eingegebenen Zahl *n* das Quadrat:

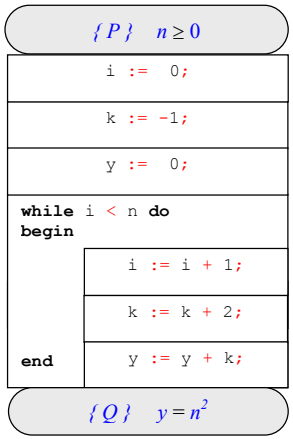
```
function quad(n : integer) : integer;
var i, k, y : integer;
begin
  i :=  0;
  k := -1;
  y :=  0;
  while i < n do
  begin
    i := i + 1;
    k := k + 2;
    y := y + k;
  end;
  quad := y;
end;
```

Die für die Verifikation relevanten Teile dieses Quelltextes (grau) und die zu behandelnden Abschnitte (farbige Unterteilungen):

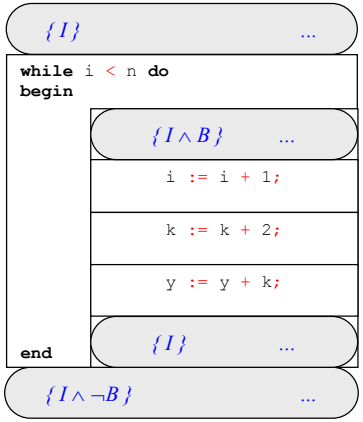
```
i := 0;
k := -1;
y := 0;
while i < n do
  i := i + 1;
  k := k + 2;
  y := y + k;
```

Es ergeben sich also sieben zu verifizierende Einheiten, also sechs Zuweisungen und eine While-Schleife, die auf zwei Ebenen verteilt sind, nämlich vor der While-Schleife und darin.

1. Die Vor- und Nachbedingungen des Quelltextes beinhalten, was für die Eingangs- und Rückgabewerte gelten soll, nämlich, dass $n \geq 0$ sein soll und dass das Ergebnis für y gleich n^2 lautet:



2. Finden der äußeren unteren Anweisung: Umfassend und zuunterst steht die While-Schleife, damit wird zuerst die Iterationsregel angewandt. Damit stehen die Positionen, an denen Hoare-Klauseln zu vermerken sind, fest:



Etwas problematischer ist in diesem Fall die Ermittlung der Invarianten. Um die Schleife zu verifizieren, muß sie verstanden werden – das mußte sie auch, um überhaupt programmiert werden zu können. Diese Gedanken gilt es nachzuvollziehen. Ziel der Überlegung ist herauszufinden, wie die vier Variablen i, k, y und n miteinander in Beziehung stehen. Hilfreich ist hierbei häufig das Anlegen einer Tabelle mit den Variablen sowie deren Belegung mit einigen Werten, so lassen sich Gesetzmäßigkeiten einfacher erkennen:

Variablen	i	k	y	n
Schleifen-Eintritt	0	-1	0	5
Erster Durchlauf	1	1	1	5
Zweiter Durchlauf	2	3	4	5
Dritter Durchlauf	3	5	9	5
Vierter Durchlauf	4	7	16	5
Fünfter Durchlauf	5	9	25	5

Neben den Zusammenhängen, die aus dem Quelltext direkt folgen (wie etwa $y := y + k$) läßt sich nun sehen, daß k immer dem Doppelten von i subtrahiert um 1 entspricht:

$$k := 2\,i - 1$$

Der Vergleich der Spalten i ergibt den Zusammenhang, das i^2 immer y ergibt:

$$y := i^2$$

Ferner ist i immer kleiner oder gleich n:

$$i \leq n$$

Alle diese Eigenschaften treffen an den für die Invariante I geforderten Positionen zu, so daß sie versehen mit einer Konjunktion übernommen werden können:

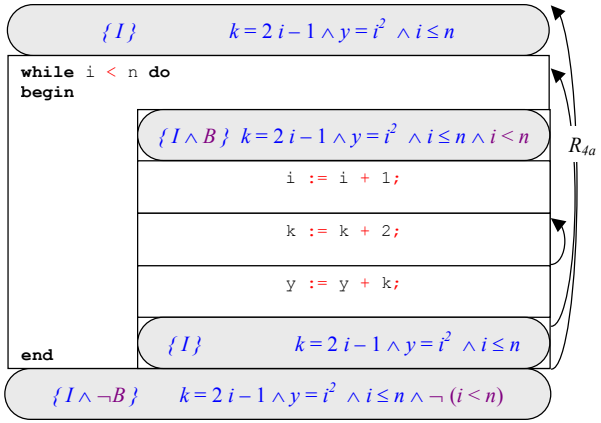
$$I:\ k = 2\,i - 1 \ \wedge\ y = i^2 \ \wedge\ i \le n$$

Offen sind nun noch B und $\neg B$, die sich durch Ablesen bzw. durch Negieren des Abgelesenen ergeben:

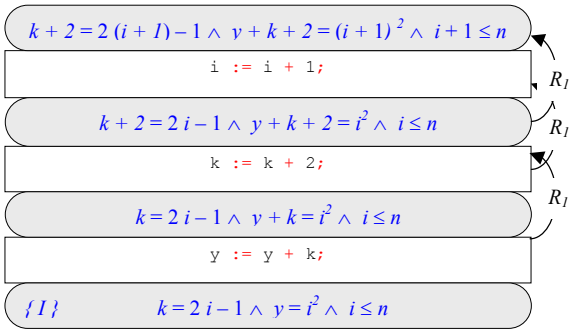
$$B:\quad i < n$$

$$\neg B:\ \neg\,(i < n)$$

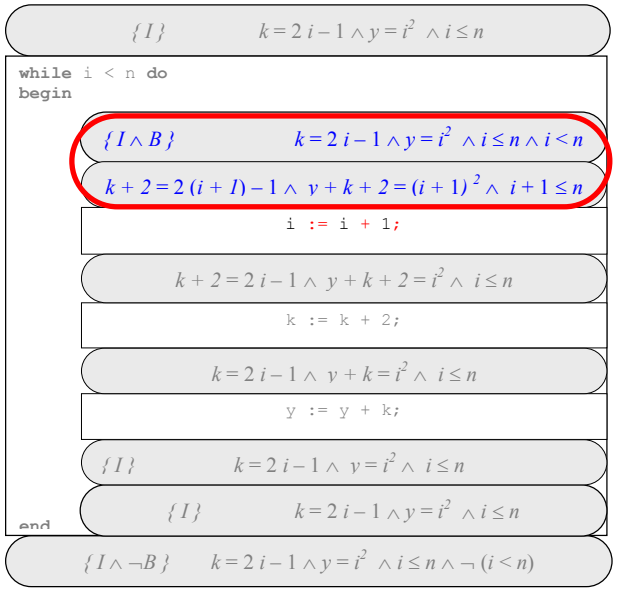
Damit ergibt sich:



3./4.Da es im Falle der While-Schleife innere Anweisungen (die drei Zuweisungen) gibt, sind diese zunächst mittels Zuweisungsaxiom von unten nach oben zu verifizieren:



Die oberste Klausel dieses Abschnitts trifft nun auf die zweite Klausel $I \wedge B$, die für die Iterationsregel aufgestellt wurde. Damit die Verifikation fortgesetzt werden kann, müssen die augenscheinlich nun differierenden Terme in den Klauseln abgeglichen werden[5].



Es ergibt sich also:

$$\underbrace{k+2=2(i+1)-1}_{\text{Teil 1}} \wedge \underbrace{y+k+2=(i+1)^2}_{\text{Teil 2}} \wedge \underbrace{i+1\le n}_{\text{Teil 3}} \Leftrightarrow \underbrace{k=2i-1}_{\text{Teil 1}} \wedge \underbrace{y=i^2}_{\text{Teil 2}} \wedge \underbrace{i\le n \wedge i<n}_{\text{Teil 3}}$$

Die einzelnen durch Konjunktionen verbundenen Teilterme der linken und rechten Seite werden nun separat überprüft.

Teil 1:

$$\begin{aligned} & k + 2 = 2\,(i + 1) - 1 \quad / \text{-}2 \\ \Leftrightarrow\ & k = 2 \cdot i + 2 - 1 \\ \Leftrightarrow\ & k = 2 \cdot i - 1 \end{aligned}$$

Teil 2:

$$\begin{aligned} & y + k + 2 = (i + 1)^2 \\ \Leftrightarrow\ & y + k + 2 = i^2 + 2\,i + 1 \quad / \text{-}2 \\ \Leftrightarrow\ & y + k = i^2 + 2\,i - 1 \\ \Leftrightarrow\ & y = i^2 \ \wedge\ k = 2 \cdot i - 1 \end{aligned}$$

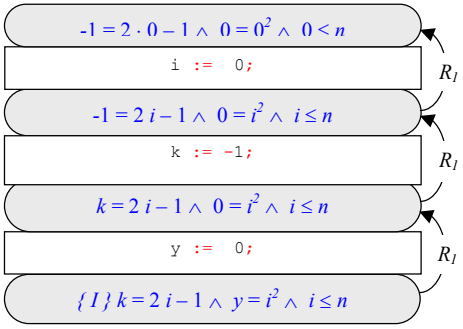
[5] Ist dies nicht machbar, kann relativ sicher von einem gefundenen Fehler ausgegangen werden, der sicherlich auch für den Ablauf des Programms nicht unerheblich wäre. Das Finden und Eliminieren solcher Fehler ist auch Sinn der Verifikation, schließlich entspricht es dem Stand der Dinge, daß Programme nicht fehlerfrei sind.

Teil 3:

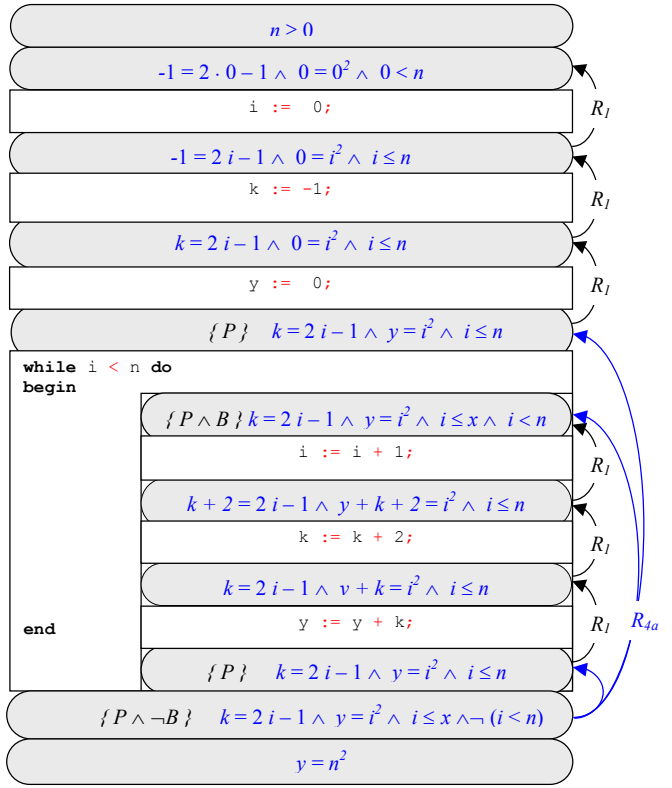
$i + 1 \le n \quad = \quad i < n$

Die Bedingungen $i \le n$ und $i < n$ stellen keinen Widerspruch zueinander dar, die zweite Bedingung formuliert lediglich denselben Sachverhalt schärfer und ist daher maßgeblich. Insgesamt lassen sich die zunächst unterschiedlich erscheinenden Aussagen zur Deckung bringen, die Verifikation kann fortgesetzt werden.

5. Abschließend werden die drei Zuweisungen über der While-Schleife mit Klauseln versehen. Ausgangsbasis hierfür ist die Invariante der While-Schleife, damit wird widerum von unten nach oben gearbeitet.



Insgesamt ergibt sich die folgende Spezifikation:



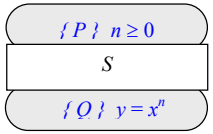
3.6.3 Bilden des Exponentiation aus einer Basis und einem Exponenten

Das folgende Programm bildet aus zwei gegebenen Zahlen x und n mit n ≥ 0 das Ergebnis x^n:

```
function exp (x : integer; n : integer) : integer;
var k, p, y : integer;
begin
  k := n;
  p := x;
  y := 1;
  while k > 0 do
  begin
    if (k mod 2 = 0)
    then begin
           p := p * p;
           k := k div 2;
         end
    else begin
           y := y * p;
           k := k - 1;
         end;
  end;
  exp := y;
end;
```

Anweisungen, die zu verifizieren sind

Es ergibt sich damit folgender formale Rahmen:



Liegen eindeutige Einrückungen[6] vor, kann die unterste umfassende Anweisung schnell gefunden werden. Im obigen Quelltext ist die While-Schleife von den zu verifizierenden Anweisungen die sich am weitesten links unten befindliche. Durch die grüne Einrahmung herausgestellt aber auch in der Struktogramm-Darstellung erkennbar, befindet sich die If-Verzweigung innerhalb der Schleife und wird somit erst nach dem While-Konstrukt verifiziert.

Verifikation der While-Schleife

Für die Verifikation der While-Schleife werden die Zustandsbeschreibungen I, B und $\neg B$ benötigt, wobei I eine Invariante und B die Schleifenbedingung darstellt. Die Schleifenbedingung B geht aus dem Quelltext hervor:

$$B: \quad k > 0$$
$$\neg B: \quad \neg(k > 0)$$

Finden einer Invarianten

Für das Finden einer Invarianten ist ein tiefgehenderes Verständnis der Abläufe in der Schleife notwendig, hierbei handelt es sich um einen kreativen und bislang nicht automatisierbaren Prozess. Zum Verständnis der Schleife hilfreich ist häufig das Anfertigen einer Tabelle und das Belegen der Variablen mit verschiedenen aufsteigenden Werten, um Gesetzmäßigkeiten zu erkennen. Zur besseren Sichtbarkeit wird x ‘stillgehalten‘ d. h. nicht wie k mit verschiedenen Werten belegt sondern symbolisch behandelt. Für n wird der Wert 10 vorgegeben.

[6] *Jedes* **begin** hat einen vorhergehenden Zeilenumbruch, der folgende Quellcode ist eine bestimmte Anzahl von Leerzeichen eingerückt (mindestens um ein Leerzeichen) und auf einer Einrückungshöhe mit dem **begin** befindet sich das zugehörige **end** (so daß der Cursor zwischen dem Anfang von **begin** und **end** bündig hin- und herbewegt werden kann).

Tabelle 1, Tabelle mit symbolischer Auswertung der Zustände von y, p *und* k. *Die grauen Eintragungen markieren die im aktuellen Durchlauf nicht veränderten Größen.*

Variablen	*y*	*p*	*k*	*x*	*N*
Schleifen-Eintritt	1	x	10	x	10
Erster Durchlauf	1	x^2	5	x	10
Zweiter Durchlauf	x^2	x^2	4	x	10
Dritter Durchlauf	x^2	x^4	2	x	10
Vierter Durchlauf	x^2	x^8	1	x	10
Fünfter Durchlauf	x^{10}	x^8	0	x	10

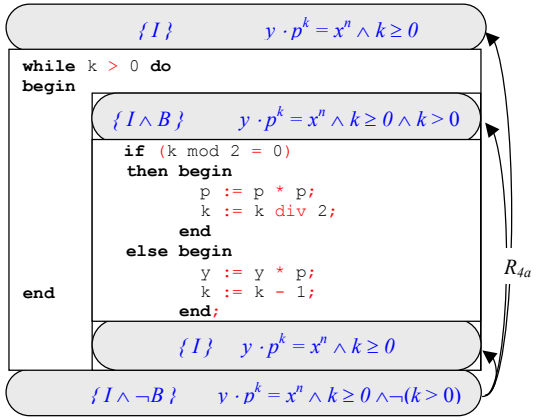
Damit ergibt sich für die Invariante der zunächst die Gesetzmäßigkeit $x^n = y \cdot p^k$, ferner gilt innerhalb der Schleife immer, daß $k \geq 0$ ist. Somit gilt

$$I: \qquad x^n = y \cdot p^k \quad \wedge \quad k \geq 0$$

Iterationsregel:
- $\{ I \}$,
- $\{ I \wedge B \}$,
- $\{ I \wedge \neg B \}$

müssen gefunden werden

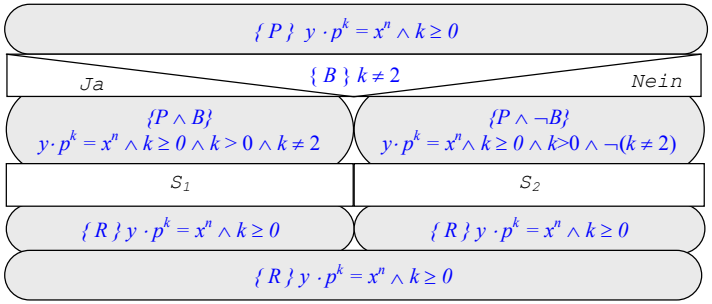
Eintragen von I, B und $\neg B$ führt auf folgende Spezifikation:



Verifikation der Verzweigung

Die If-Verzweigung ist somit von Zustandsbeschreibungen umgeben ($I \wedge B$ oben sowie I unten) die hier als Vorbedingung P und Nachbedingung R für die Anwendung der Regel der Fallunterscheidung übernommen werden. Die Klauseln $P \wedge B$ sowie $P \wedge \neg B$ lassen sich sehr leicht finden: B stellt die Bedingung

der Verzweigung dar und ist nur abzulesen, da P und R der jeweils äußeren Invarianten I entsprechen, ist diese ebenfalls nur zu übertragen.



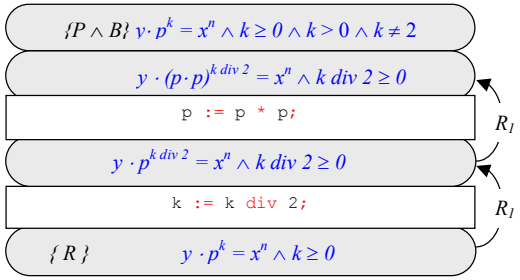
Regel der Fallunterscheidung:

- $\{ P \}$,
- $\{ B \}$,
- $\{ P \wedge B \}$,
- $\{ P \wedge \neg B \}$
- $\{ R \}$

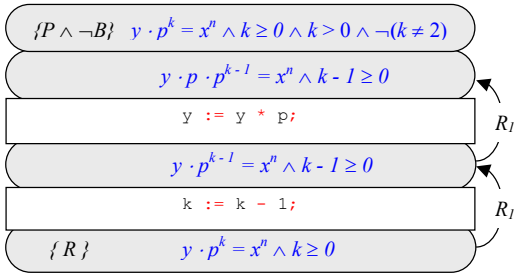
müssen gefunden werden

Damit bleiben als letzte innere Elemente die beiden Anweisungsfolgen S_1 und S_2. Beide werden nacheinander nach dem Zuweisungsaxiom mit Klauseln durchsetzt und die sich ergebenden Vorbedingungen werden mit den Nachbedingungen der umfassenden Klauseln (aus der Verzweigung) abgeglichen.

Fall 1: $k \neq 2$

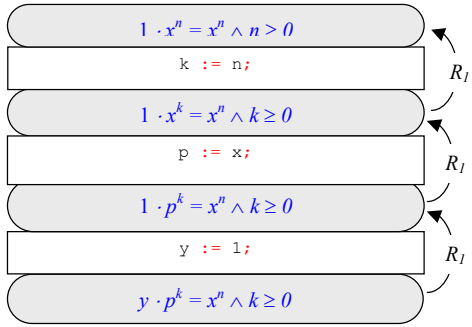


Fall 2: $\neg(k \neq 2)$

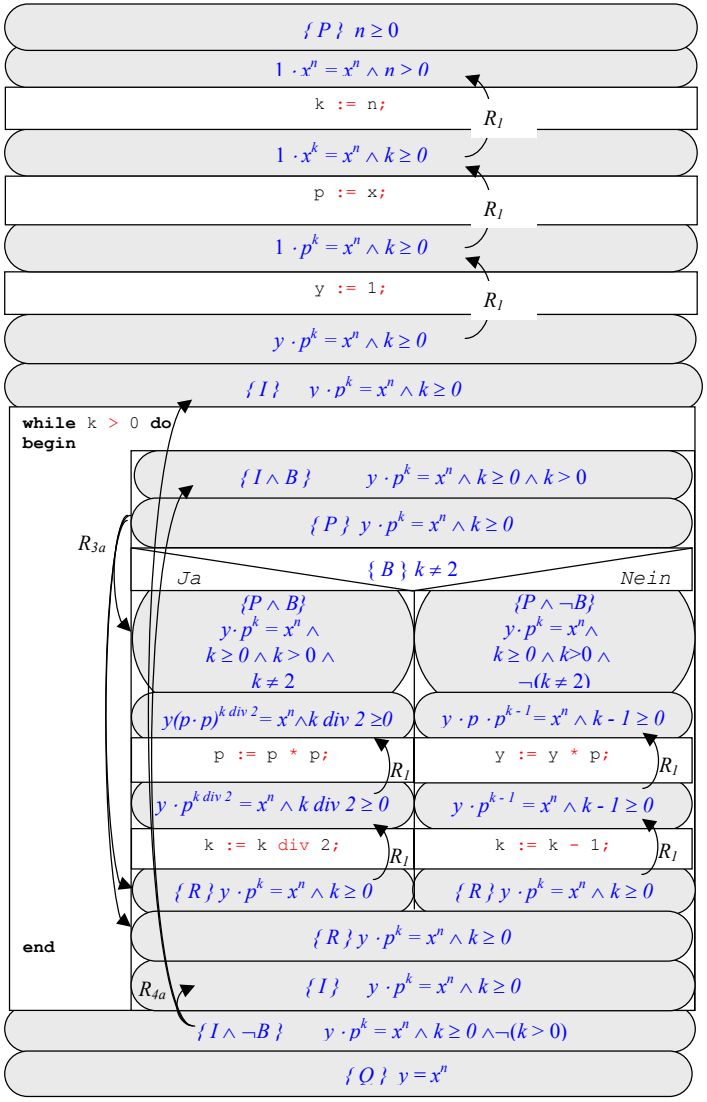


Hierbei geht alles auf. Die verschachtelten Anweisungen sind verifiziert, die Arbeit kann oberhalb der While-Schleife fortgesetzt werden. Hier befinden sich

drei weitere Zuweisungen, die durch weitere Anwendungen des Zuweisungsaxioms bearbeitet werden.



Hierbei ergeben sich ebenfalls keine Widersprüche in sich oder gegenüber der anfänglichen Vorbedingung des Quelltextes. Die Beweisskizze zusammengesetzt:



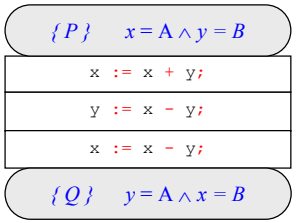
3.6.4 Verifikation von Anweisungen zum Vertauschen zweier Variablen

Die folgende Prozedur vertauscht die Inhalte zweier Variablen:

```
procedure swap(var x, y : integer) : integer;
begin
  x := x + y;
  y := x - y;
  x := x - y;
end;
```

Anweisungen, die zu verifizieren sind

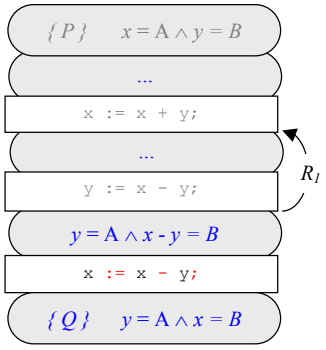
Als Erstes werden die Vor- und Nachbedingungen der gesammten Spezifikation eingetragen. Das Problem besteht hierbei in der Findung einer formalen Beschreibung. Dies kann durch die Einführung zweier Zustände A und B gelöst werden.



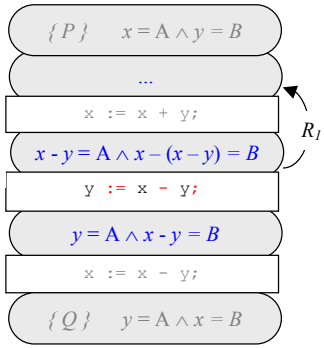
Also: Anfangs befindet sich die Variable x in einem Zustand A und die Variable y in einem Zustand B, nachdem die Anweisungen abgearbeitet sind, soll sich die Variable y in genau dem Zustand A befinden, den x am Anfang hatte und x in dem Zustand B, den y zu Beginn hatte.

Dieses Beispiel veranschaulicht insbesondere insbesondere den Umgang mit Zuweisungen, da diese Verifikation ausschließlich durch dreifache Anwendung dieses Axioms durchgeführt werden kann.

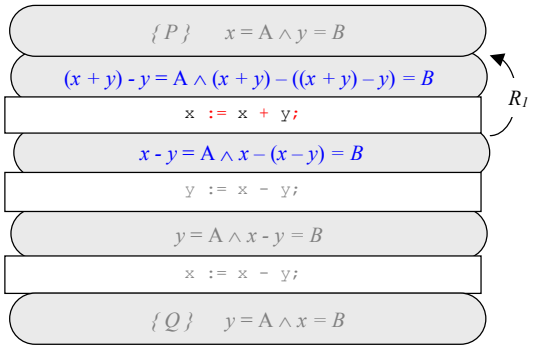
Erste Anwendung von R_1:



Zweite Anwendung:



Dritte Anwendung:



Die formale Verifikation ist abgeschlossen, die Vorbedingung P und die jetzt folgende erste Klausel müssen nun noch miteinander in Übereinstimmung gebracht werden (was durch Auflösen der Klammern und einfaches Auswerten der Terme schnell abgeschlossen werden kann):

$$\begin{aligned} x = A \wedge y = B \Leftrightarrow\ & (x + y) - y = A \wedge (x + y) - ((x + y) - y) = B \\ \Leftrightarrow\ & x + y - y = A \wedge (x + y) - (x + y - y) = B \\ \Leftrightarrow\ & x = A \wedge (x + y) - x = B \\ \Leftrightarrow\ & x = A \wedge x = B \end{aligned}$$

Die letzte Abbildung (dritte Anwendung des Zuweisungsaxioms) entspricht gleichzeitig der fertigen Spezifikation.

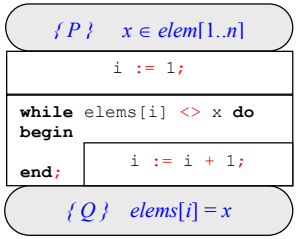
3.6.5 Verifikation einer linearen Suche

Das folgende Stück Code sucht ein enthaltenes Element x in einem Feld *elems* mit n Elementen, indem von vorne bis hinten durch das Feld gegangen und das aktuelle Element mit x verglichen wird:

```
function linsearch(x : value; elems : vector) : integer;
begin
  i := 1;
  while elems[i] <> x do
  begin
    i := i + 1;
  end;
  linsearch := i;
end;
```

Anweisungen, die zu verifizieren sind

Aufstellen der umfassenden Vor- und Nachbedingungen:



Verifikation der Verzweigung mit der Iterationsregel

Die erste umfassende Anweisung von unten ist die While-Schleife. Der erste Schritt besteht somit darin, eine Invariante zu finden. Dies gelingt hier lediglich über eine Überlegung, eine Tabelle läßt sich hier nicht sinnvoll bilden, die Nachbedingung kann auch nicht einfach übernommen werden. Die Idee ist folgende: Anfänglich liegt das gesuchte Element im Intervall $[1 .. n]$, in der Schleife wird i initialisiert und jeweils das Intervall $[i .. n]$ mit größer werdendem i durchsucht. Ohne eine solche minimale Idee geht es hier nicht. Damit läßt sich jedoch eine Invariante formulieren:

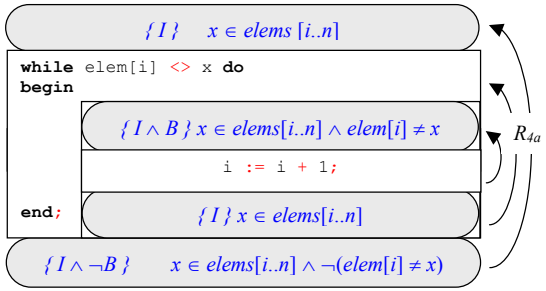
Finden einer Invarianten

$$I: x \in elems\ [i..n]$$

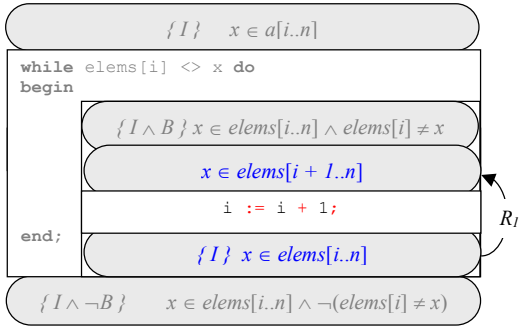
B und $\neg B$ werden wie immer durch Ablesen ermittelt, die weiteren benötigten Klauseln lauten somit:

$$\begin{aligned} I \wedge B:&\quad x \in elems\ [i..n] \wedge elems[i] \neq x \\ I \wedge \neg B:&\ x \in elems\ [i..n] \wedge \neg(elems[i] \neq x) \end{aligned}$$

Damit kann die sich aus dem Hoare-Kalkül ergebende Schablone wie folgt ausgefüllt werden.



Mit der Zuweisung in der Schleife wird gemäß dem Zuweisungsaxiom verfahren:



Hier stellt sich die Frage ob

$$x \in elems[i\,..\,n] \wedge elem[i] \neq x \Leftrightarrow x \in elems[i+1\,..\,n]$$

gilt. Dies läßt sich nur über eine durchaus anschauliche Logik erschließen.

$$x \in elems[i+1\,..\,n] = x \notin elems[1\,..\,i]$$

Abgleichen der Zwischenergebnisse

Also: Wenn x element einer Menge $[i + 1 .. n]$ ist, kann es nicht mehr Element einer (durchsuchten) Menge $[1 .. i]$ sein, also auch nicht $elems[i]$. Tatsächlich folgt hierraus die rechte Teilaussage, daß

$$elem[i] \neq x$$

muß. Der linke Teil stellt ebenfalls keinen Widerspruch dar, wenn

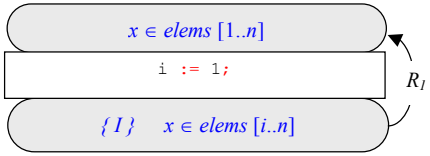
$$x \in elems[i+1\,..\,n]$$

gilt, dann gilt natürlich auch

$$x \in elems[i\,..\,n]$$

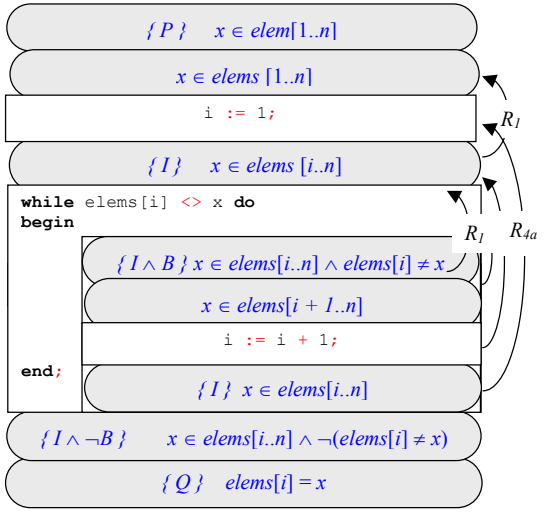
Hier ist lediglich ein Element zuviel in der Menge, enthalten bleibt das gesuchte Element natürlich dennoch.

Zuletzt bleibt noch die Zuweisung oberhalb der While-Schleife, die wiederum mit dem Zuweisungsaxiom bearbeitet wird. Die Nachbedingung der Zuweisung wird von der While-Schleife übernommen (die letzte Klausel unter der zu bearbeitenden Zuweisung ist die Schleifeninvariante). Eine einfache Anwendung führt auf folgende Spezifikation:



Dies entspricht der Vorbedingung, die Verifikation ist damit abgeschlossen.

Die gesamte Beweisskizze:



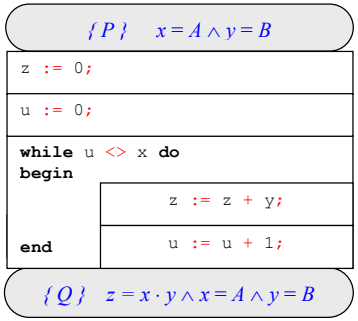
3.6.6 Verifikation einer Multiplikation durch Aufsummierung

Die hier durchgeführte Multiplikation für $z = x \cdot y$ ist nichts anderes als die x-malige Aufsummierung von y, also $z = y + y + y + ...$ (x-mal)

```
function mult(x, y : integer): integer;
begin
  z := 0;
  u := 0;
  while u <> x do
  begin
    z := z + y;
    u := u + 1;
  end;
  mult := z;
end;
```

Anweisungen, die zu verifizieren sind

1. Vor- und Nachbedingungen des gesamten Programmteils festlegen. z.B.:



2. Finden der Klauseln für die While-Schleife zur Anwendung der Iterationsregel R_{4a}. Wenn eine mögliche Invariante beim Veranschaulichen des Quelltextes nicht gesehen wird, kann in diesem Fall wieder mit einer Tabelle gearbeitet werden:

Finden einer Invarianten

Variablen	*u*	*z*	*x*	*y*
Schleifen-Eintritt	0	0	*A*	*B*
Erster Durchlauf	1	*B*	*A*	*B*
Zweiter Durchlauf	2	*B+B*	*A*	*B*
Dritter Durchlauf	3	*B+B+B*	*A*	*B*
Vierter Durchlauf	4	*B+B+B+B*	*A*	*B*
Fünfter Durchlauf	5	*B+B+B+B+B*	*A*	*B*
n-ter Durchlauf	$n = A$	$A \cdot B$	*A*	*B*

Hieran läßt eine Gesetzmäßigkeit wie

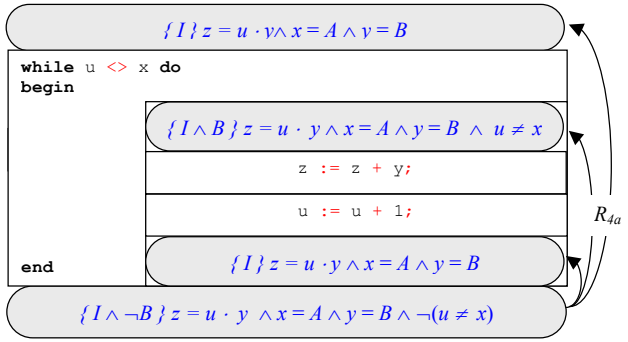
$$z = u \cdot y$$

erkennen, die sich auch als Invariante gut eignet. Mit der abzulesenden Schleifenbedingung ergibt sich für

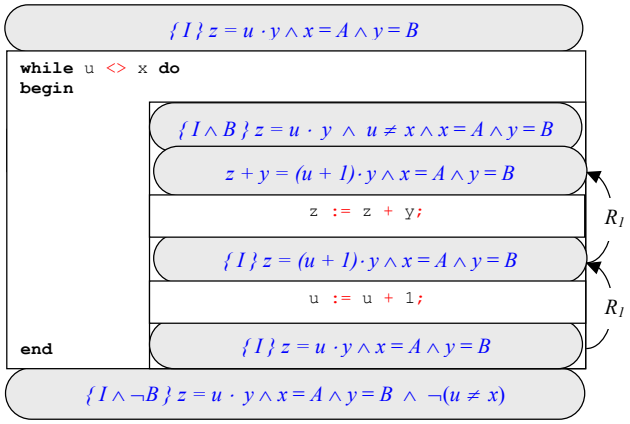
$$I \wedge B: \quad z = u \cdot y \wedge x = A \wedge y = B \wedge u \neq x$$

$$I \wedge \neg B: \quad z = u \cdot y \wedge x = A \wedge y = B \wedge \neg(u \neq x)$$

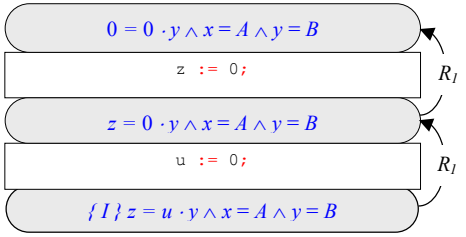
In die Spezifikation eingetragen, sieht dies wie folgt aus:



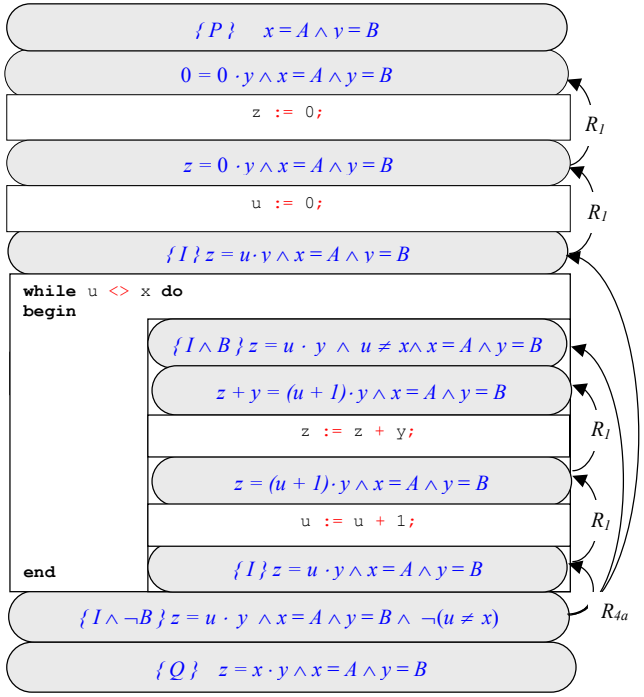
Für die Zuweisungen im Inneren der Schleife wird das Zuweisungsaxiom herangezogen.



Dem schließt sich die Abarbeitung der oberen initialen Zuweisungen an:



Dies führt auf folgende Spezifikation:



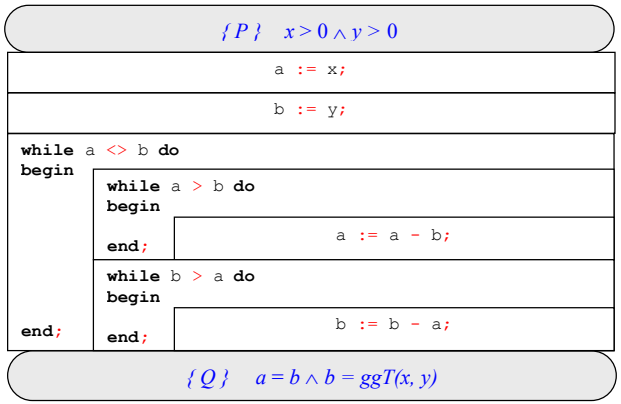
3.6.7 Verifikation von Code zur Ermittlung des ggT

Der folgende Code zieht jeweils die Variable mit dem aktuell größeren Wert von der mit dem momentan kleineren Wert ab. Dies wird wiederholt, bis beide Variablen den gleichen Wert haben, ist dies erreicht, ist der ggT ermittelt.

```
function ggT(x, y : integer) : integer;
var a, b : integer;
begin
  a := x;
  b := y;
  while a <> b do
  begin
    while a > b do
      a := a - b;
    while b > a do
      b := b - a;
  end;
  ggT := b;
end;
```

Anweisungen, die zu verifizieren sind

1. Umfassende Vor- und Nachbedingungen aufstellen:

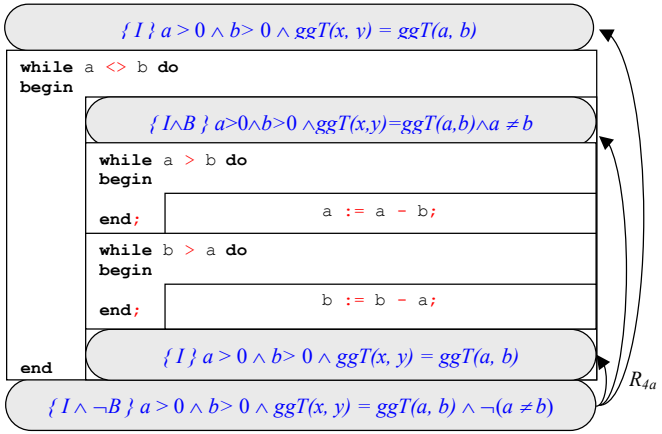


3. Mit der äußersten umfassenden Anweisung von unten beginnen. Dies ist hier die äußere While-Schleife, daher wird hier nach Regel R_{4a} vorgegangen:

Hierzu muß wieder eine Invariante gefunden werden. Hier folgt die Invariante aus einer Überlegung: ein Teiler kann nicht 0 sein, da die Division durch 0 nicht definiert ist (liegen etwa zwei Primzahlen vor, ist der größte gemeinsame Teiler 1). Des Weiteren entspricht der gesuchte Teiler *ggT(x, y)* immer *ggT(a,b)*. Hieraus wird die Invariante gebildet (die Schleifenbedingung *B* und deren Verneinung $\neg B$):

$I \wedge B$: $a > 0 \wedge b > 0 \wedge ggT(x, y) = ggT(a, b) \wedge a \neq b$

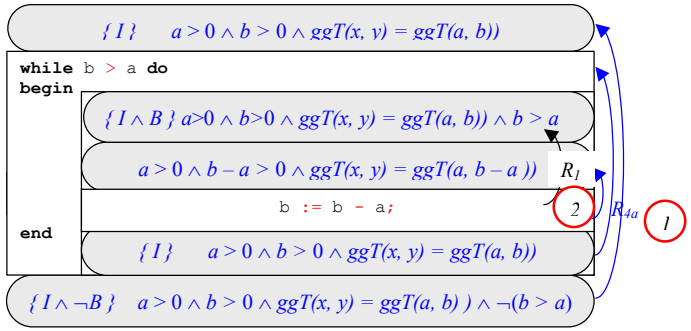
$I \wedge \neg B$: $a > 0 \wedge b > 0 \wedge ggT(x, y) = ggT(a, b) \wedge \neg(a \neq b)$



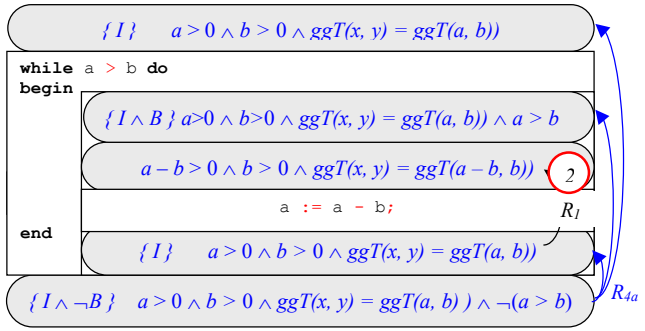
Iterationsregel:
- *{ I },*
- *{ I ∧ B },*
- *{ I ∧ ¬B }*

müssen gefunden werden

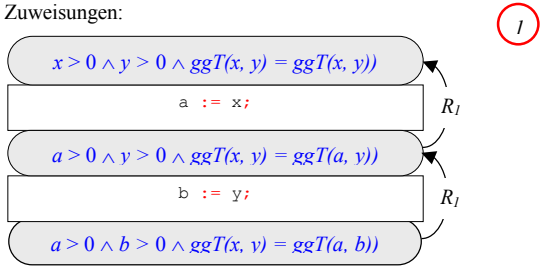
4. Von unten nach oben werden nun die weiteren Anweisungen mit Hoare-Klauseln versehen. Die Überlegungen, die auch schon bei der ersten While-Schleife getroffen wurden, erweisen sich auch bei der zweiten Schleife als gültig. Im folgenden sind sowohl die Schleifen als auch die enthaltenen Zuweisungen verifiziert.



Die letzte innere While-Schleife (mit derselben Invariante):



Die oberen Zuweisungen:



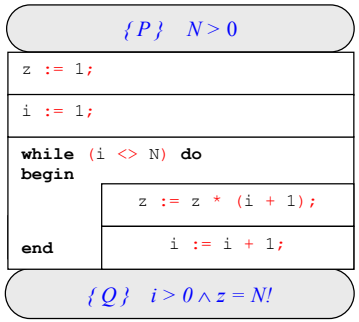
3.6.8 Verifikation von Code zur Berechnung der Fakultät einer Zahl

Dieser Code multipliziert eine aufsteigende Folge natürlicher Zahlen von 2 bis N und speichert das jeweilige Ergebnis in der Variablen z. Nach Terminierung des Codefragments, bleibt die Fakultät von N in z erhalten.

```
function fac (N : integer) : integer;
var i, z : integer;
begin
  z := 1;
  i := 1;
  while (i <> N) do
  begin
    z := z * (i + 1);
    i := i + 1;
  end;
  fac := z;
end;
```

Anweisungen, die zu verifizieren sind

1. Festlegen der Vor- und Nachbedingungen



2. While-Schleife, Bestimmung einer Invariante

Variablen	*z*	*i*
Schleifen-Eintritt	1	1
Erster Durchlauf	2	2
Zweiter Durchlauf	6	3
Dritter Durchlauf	24	4
Vierter Durchlauf	120	5

Iterationsregel:
- *{ I }*
- *{ I ∧ ¬B }*
- *{ I ∧ B }*

müssen gefunden werden

Hieraus ist zu ersehen, daß zwischen z und i daß Verhältnis

$$z = i!$$

bzw.

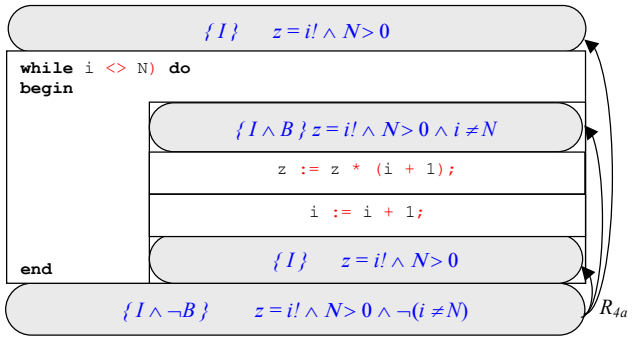
$$z = \prod i$$

gilt. Ferner ist vorauszusetzen, daß N größer als 0 ist, da die Schleifenbedingung sonst nie wahr werden kann und i bis zu einem Überlauf oder einer Exception zählt. Mit diesen beiden Regeln kann eine Invariante ausgedrückt werden, mit der Schleifenbedingung (ablesen bzw. ablesen und negieren) ergibt sich:

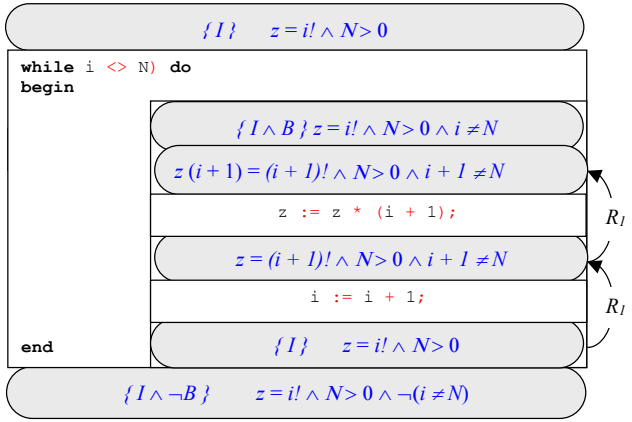
$$I \wedge B: \quad z = i! \wedge N > 0 \wedge \quad i \neq N$$

$$I \wedge \neg B: z = i! \wedge N > 0 \wedge \neg(i \neq N)$$

Eingesetzt in die Iterationsregel:



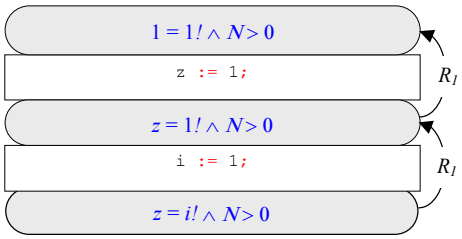
Spezifikation der Anweisungen innerhalb der Schleife:



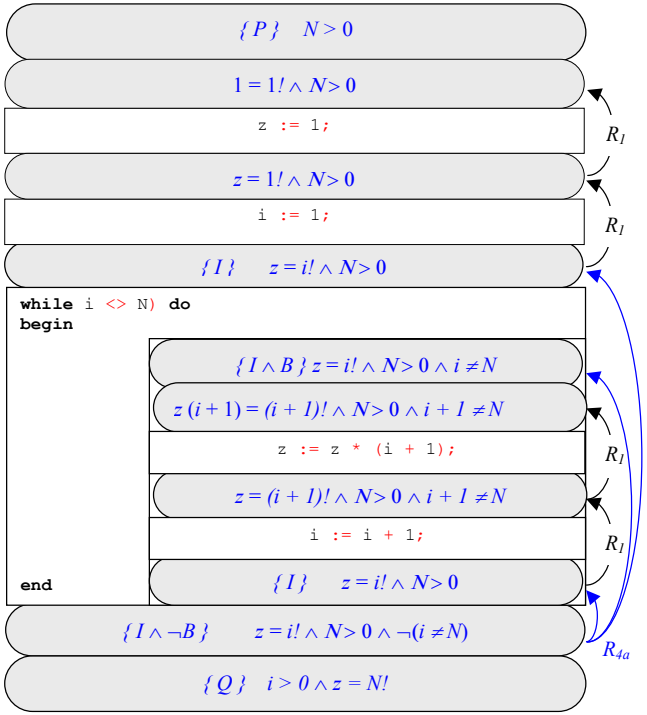
Dies läßt sich jedoch auflösen:

$$\begin{aligned} z\,(i+1) &= (i+1)! \\ &= (i+1) \cdot i! \quad / (i+1) \\ z &= i! \end{aligned}$$

Damit bleiben die initialen Anweisungen vor der While-Schleife:



Dies bestätigt die anfänglichen Voraussetzungen. Dies gesamte Lösungsskizze:



3.6.9 Verifikation eines Programms zur Bestimmung der Prim-Eigenschaft einer Zahl

Das folgende Programm ermittelt zu einer Zahl x, ob sie eine Primzahl ist oder nicht und gibt dies in Form eines Wahrheitswertes zurück. Hierzu wird eine Variable von 2 bis x inkrementiert und bei jedem Schritt geprüft, ob *x* durch diese Variable geteilt werden kann. Es wird also mit Brute-Force jede Zahl durch *x* geteilt und hierbei mit der Abfrage

```
x mod i <> 0
```

getestet, ob ein Rest bleibt. Läßt sich eine Zahl, die kleiner als *x* ist, durch *x* teilen, ergibt sich 0, es bleibt also kein Rest und *x* ist per Definition keine Primzahl.[7]

[7] Eine Primzahl ist nur durch 1 und durch sich selbst teilbar.

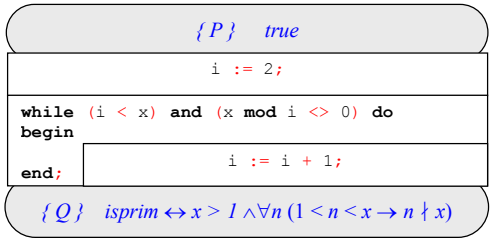
```
function isprim(x : integer) : boolean;
var i : integer;
begin
  i := 2;
  while (i < x) and (x mod i <> 0) do
  begin
    i := i + 1;
  end;
  isprim := i = x;
end;
```

Anweisungen, die zu verifizieren sind

Dies ergibt die einfache Vorbedingung *true* und die Nachbedingungen, daß der Rückgabewert *isprim* äquivalent zu den Aussagen ist, daß $x > 1$ gilt und gleichzeitig daß es keine Zahl n zwischen 1 und x gibt, durch die x teilbar ist[8]. Letzeres formal notiert:

$$isprim \leftrightarrow x > 1 \wedge \forall n\,(1 < n < x \rightarrow n \nmid x)$$

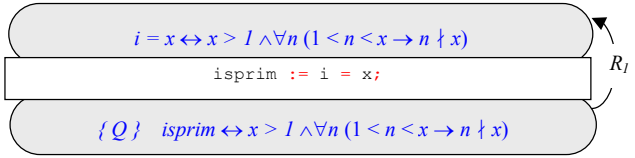
Damit kann folgendes notiert werden:



Iterationsregel:
- $\{ I \}$
- $\{ I \wedge \neg B \}$
- $\{ I \wedge B \}$

müssen gefunden werden

Zuerst die abschließende Zuweisung:



While-Schleife: Die Invariante kann durch Überlegungen aus der Nachbedingung des Programms abgeleitet werden. Immer gültig und durch den Quelltext festgelegt ist die Tatsache, daß i einen Wert zwischen 1 und x hat. Des weiteren gilt für alle Zahlen n, die die Schleife bereits durchlaufen hat, (also zwischen 1 und i) sicher, daß x nicht durch diese teilbar ist, sowie daß i größer als 1 sein muss. Dies in mathematischer Notation ausgedrückt, ergibt für die Invariante:

$$I\text{: } i > 1 \wedge \forall n\,(1 < n < i \rightarrow n \nmid x)$$

(für alle Zahlen n zwischen 1 und i gilt, daß x nicht durch sie teilbar ist)

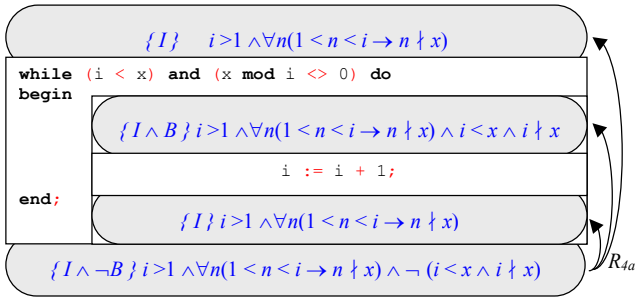
[8] Die Aussage „x ist nicht teilbar durch n“ wird durch das Symbol $\nmid$ wie folgt dargestellt: $x \nmid n$

Ferner gilt für die weiteren Klauseln (Ablesen der Schleifenbedingung bzw. Ablesen + Negieren der Schleifenbedingung für die untere Klausel):

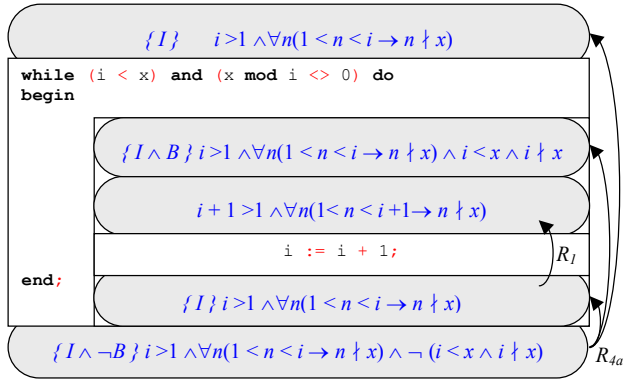
$$I \wedge B\text{: } \; i > 1 \wedge \forall n\,(1 < n < i \rightarrow n \nmid x) \wedge \quad i < x \wedge i \nmid x$$

$$I \wedge \neg B\text{: } i > 1 \wedge \forall n\,(1 < n < i \rightarrow n \nmid x) \wedge \neg\,(i < x \wedge i \nmid x)$$

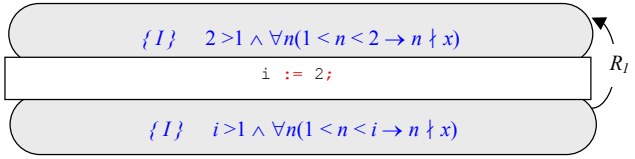
Einsetzen:



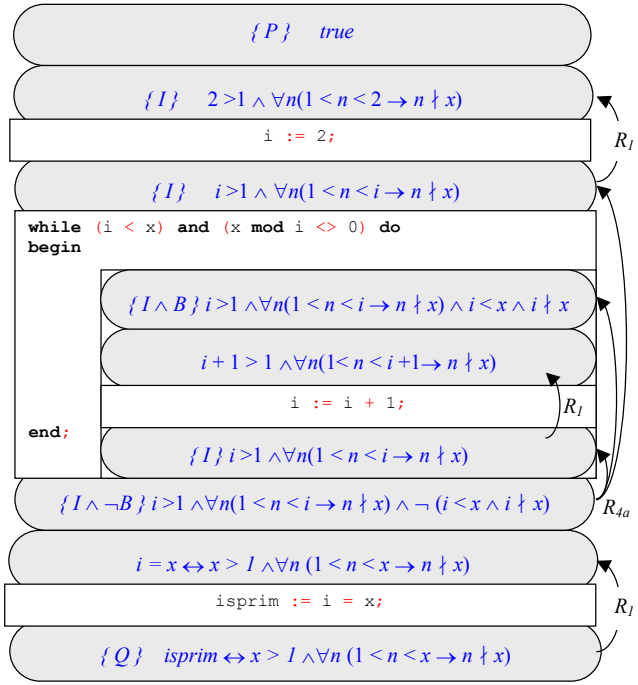
Die Zuweisung im Inneren der Schleife:



Abschließend die initiale Zuweisung:



Die gesamte Spezifikation:



4 Einschränkungen

Ein absolut nicht funktionierender Beweis nach diesem Schema ist ein guter Hinweis darauf, daß ein Programm bestimmte Unschlüssigkeiten aufweist. Allerdings ist auch das Gegenteil durchaus denkbar: Es ist ein tatsächlich fehlerfreies Stück Code entstanden, der Autor ist aber zur Beweisführung - die eben nicht immer trivial ist - nicht imstande. So läßt sich auch hier aus dem Mangel des Beweises nicht handfest schließen, sondern wirklich nur aus der Tatsache, daß Programm und Beweis vorliegen.

Für größere Programme gesellt sich die Komplexität zu den üblichen Unwägsamkeiten hinzu. Ein wirklich komplexes Stück längeren und 'hartverdrahteten' Codes - so etwas gibt es gelegentlich eben doch - verlangt einen ungleich komplexeren Beweis, bei dessen Führung mit mehr Fehlern zu rechnen ist, als das Programm bereits enthält. Anbetrachts der Tatsache das Menschen Fehler machen - und zwar je mehr, umso komplexer eine Angelegenheit wird (soviel zählt zu den allgemein gesicherten Erkenntnissen) - besteht absolut kein Grund einem solchen Beweis mehr Vertrauen zu schenken, als dem Programm, dessen Korrektheit man möglichweise bezweifelt.

Kein auf dem Programmcode aufsetzendes System wie etwa der hier vermittelte Hoare-Kalkül (auf den Ur-Informatiker *C.A.R. Hoare* wird diese Tech-

nik zurückgeführt) kann Fehler entdecken, die bereits in der Spezifikation verankert sind. Es wird maximal gezeigt, daß eine Spezifikation korrekt umgesetzt worden ist, möglicherweise eben auch eine fehlerhafte. Ein Programm mit korrekt umgesetzten Routinen, die jedoch wichtige Zustände der Wirklichkeit nicht situationsgerecht erfassen, hat in der richtigen Situation jedoch das Potential genauso gefährlich zu sein, wie eine schlechte Implementierung einer an sich sauberen Spezifikation.

Weitere Quellen

Verifikation

Gregor Engelmeier: Programmverifikation "Schritt für Schritt" (Postscript)

Literatur

[Bac 89] Backhouse, Programmkonstruktion und Verifikation. Carl Hanser Verlag und Prentice-Hall International, München, Wien, 1989.
Ein Lehrbuch, dass in die Verifikation und deren Voraussetzungen wie Aussagen- und Prädikatenlogik exzellent einführt und umfangreiche Übungen mit Lösungen bereitstellt. Manchmal wünschte man es sich vielleicht noch etwas eingängiger und didaktischer. Einiges erschließt sich erst durch „richtiges" Lesen ganzer Kapitel, da der Autor weit ausholt. Anderseits: Wenn man ein Kapitel hier richtig liest (bei Lücken eben auch noch das davor ...), hat der Autor dafür gesorgt, dass die Materie – soweit möglich – dann auch verstanden werden kann und nicht drei Seiten ein unerläuterter Formalismus lauert, über den man dann doch nicht hinauskommt. Das ist bei diesem Thema nicht unbedingt selbstverständlich.

[Boo 54], Boole, G., *Investigation of the Laws of Thought*.
Allenfalls historisch interessant.

[Cla 79] Clarke, E.M., Programming language constructs for which it is impossible to obtain good Hoare axiom systems. Journal of the ACM, 26(1), S. 129 – 147, 1979.
Allenfalls historisch interessant.

[Dij 82] Dijkstra, E. W., *Selected Writings on Computing*. Springer-Verlag, New York, 1982.
Allenfalls historisch interessant.

[Flo 67] Floyd, R., *Assigning Meaning to Programs*. In: Schwartz, J.P. [Hrsg.], Proceedings of Symposium of Applied Mathematics 19, Mathematical Aspects of Computer Science, S. 19 – 32, American Mathematical Society, New York, 1967.
Allenfalls historisch interessant.

[Hoa 69] Hoare, C.A.R., *An axiomatic basis for computer programming*. Communications of the ACM, Vol. 12, S. 567 - 583, 1969

Der Originalaufsatz von Hoare, in dem die Technik, die heute als Hoare-Kalkül bekannt ist, vorgestellt wurde. Didaktisch wohl eher zu abzulehnen, wer lediglich eine Klausur darüber bestehen will, kann diese Quelle meiden, wer hingegen an der Verifikation großes Interesse hat und damit wirklich arbeiten will, muss sich das Papier sicher einmal ansehen.

[HM 73] Hoare C.A.R., Wirth N.: *An Axiomatic Definition of the Programming Language Pascal*. Acta Informatica 2,S. 335-355, 1973.

Allenfalls historisch interessant.

[Lau 71] Lauer, P.E., Consistent formal theories of the semantics of programming languages. Technical Report 25.121, IBM Laboratory Vienna, 1971.

Allenfalls historisch interessant.

[Lov 98] Loviscach, J., Absturzgefahr, die Bug-Story. In: c't, Magazin für Computertechnik, Nr. 18/1998, S. 156 - 165.

Enthält nichts zum Hoare-Kalkül, ist allerdings eine phantastische Quelle zur Motivation. Schöne Beispiele bekannter Software-Fehler. Dies oder Ähnliches sollte man als Software-Entwickler schon einmal gelesen haben.

[Sch 95] Schöning, U., *Logik für Informatiker*. Spektrum Akademischer Verlag GmbH, Heidelberg et al., 1995.

Enthält nichts zum Hoare-Kalkül, führt aber tief in die verschiedenen Logiken ein, derer sich Beweise, die Verifikation auch und der Hoare-Kalkül im Speziellen bedienen. Für Nicht-Informatiker, die das Denken im Formalen nicht gewohnt sind, gewöhnungsbedürftig, vielleicht auch kaum verständlich (man beachte daher den Titel). Für Informatiker wohl verstehbar, aber mit Sicherheit auch nicht immer einfach.

[Tur 49] Turing, A.M., *On checking a large Routine*. Report of a Conference on High Speed Automatic Calculating Machines, S. 67 – 69, Univ. Math Laboratory, Cambridge, 1949.

Allenfalls historisch interessant.

[Wir 71] Wirth, N., *The Programming Language Pascal*. Acta Informatica 1(6), S. 35-63, 1971.

Allenfalls historisch interessant, für die Verifikation irrelevant.